Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers

Bjørn Bugge Grathwohl Fritz Henglein
Ulrik Terp Rasmussen Kristoffer Aalund Søholm
Sebastian Paaske Tørholm
DIKU, University of Copenhagen
{bugge,henglein,dolle,soeholm,sebbe}@diku.dk

September 18th, 2015

Abstract

We present and illustrate Kleenex, a language for expressing general nondeterministic finite transducers, and its novel compilation to streaming string transducers with essentially optimal streaming behavior, worst-case linear-time performance and sustained high throughput. In use cases it achieves consistently high throughput rates around the 1 Gbps range on stock hardware, performing well, especially in complex use cases, in comparison to both specialized and related tools such as `AWK`, `sed`, `grep`, RE2, Ragel and regular-expression libraries.

1 Introduction

Imagine you want to implement syntax highlighting. This can be thought of as parsing the input into its tokens and processing each token according to its class. For illustration, assume we have one keyword, `for`, and alphabetic identifiers as the only tokens. The lexical structure of the input is essentially described by the *regular expression (RE)* $((\texttt{for}|[a-z]^*)\ )^*$, where whitespace is, for simplicity, represented by the single blank between the two closing parentheses. This scenario highlights the following:

Ambiguity by design. The RE is ambiguous. The intended semantics is that the left alternative has higher priority than the right. This is *greedy* disambiguation: Choose the left alternative if possible, treating e^* as its *unfolding* $ee^*|1$. Accordingly, in our example `for` matches the left alternative, not the right.

Regular expression parsing. Note that the RE has star height 2; in particular, we need to *parse* the input under multiple Kleene stars. For

our RE the parse of a string is a list of segments (corresponding to the outer Kleene star), with each segment represented by a pair, a token and whitespace (corresponding to concatenation), where each token is tagged (corresponding to the alternation) to indicate that it is either the keyword `for` or an identifier; an identifier, in turn, consists of a list (corresponding to the inner Kleene star) of characters.

Output actions. We need to output something, the highlighted tokens, not just *accept* or *reject* a string as is done by finite *automata*. Note that output actions are not specified in our RE.

We would like to do the highlighting in a *streaming* fashion, using as little internal storage as possible and performing output actions as early as they are determined by the input prefix read so far, at a high sustained input processing rate, in particular in worst-case linear-time in the length of the input stream with a low factor depending linearly on the size of the RE. We would like to accomplish this automatically for *arbitrary* REs (or similar input format specification) and output actions, with speeds that in practice adapt to how much output actually needs to be produced; in particular, performance should gracefully approach pure acceptance testing as more and more output actions are removed. How?

It turns out that the set of parses are exactly the elements of the RE read as a *type* [**?**, **?**]: Kleene-star is the (finite) list type constructor, concatenation the Cartesian product, alternation the sum type and an individual character the singleton type containing that character. A *Thompson automaton* [**?**] represents an RE in a strong sense: the complete paths—paths from initial to final state—are in one-to-one correspondence with the parses [**?**]. If a string has 4 parses (e.g. "for for "), then there are exactly 4 complete paths accepting it. Let us look at bit closer at a Thompson automaton: It is nondeterministic, with ϵ-transitions, easily constructed, having $O(m)$ states and transitions from an RE of size m. It has exactly one initial and one accepting node. Every state is either *nondeterministic*: it has two outgoing ϵ-transitions ("left" or "right"); or it is *deterministic*: it has exactly one outgoing transition labeled by ϵ or an input symbol, or it is the final state, which has no outgoing transition. Every complete path is determined by a sequence of bits used as an oracle [**?**]. Starting with the initial state, follow all outgoing transitions from deterministic states; upon arriving at a nondeterministic state query the oracle to determine whether to go left or right, until the final state is reached. The bit sequence of query responses yields a prefix-free binary code for the string accepted on the designated path. This *bit-code* can also be computed directly from the RE underlying the Thompson NFA [**?**, **?**]. Since a bit-code represents a particular parse, a string can have multiple bit-codes if and only if the RE (and thus Thompson automaton) is ambiguous: The *greedy* parse of a string, which we are interested in, corresponds to the *lexicographically least* amongst its bit-codes [**?**].

The *greedy RE parsing problem* is producing this lexicographically least bit-code for a string matching a given RE. This can be done by an *optimally streaming* algorithm, running in time linear in the size of the input string for fixed RE [**?**]: The bits in the output are produced as soon as they are uniquely determined by the input prefix read so far, assuming the input string will eventually be accepted. The algorithm maintains an ordered *path tree* from the initial state to all the automata states reachable by the input prefix read so far. A branching node represents both sides of an alternation that are both still viable. The (possibly empty) path segment from the initial state to the first branching node is what can be output based on the input prefix processed so far, without knowing which of the presently reached states will eventually accept the rest of the input. This works for all REs and all inputs; e.g., it automatically results in constant memory space consumption for REs that are deterministic modulo finite look-ahead, e.g. one-unambiguous REs [**?**].

Let us step back a bit. It is possible to aggressively ("earliest possible") and efficiently stream out the bit-code of the greedy parse of an input string under a given RE as the input is streaming in: worst-case linear time in the input string size, no backtracking and each input symbol can be processed in time $O(m)$, linear in the size of the RE and of its Thompson NFA. (Here it is critical that Thompson NFAs have ϵ-transitions since equivalent ϵ-free automata require $\Omega(m \log m)$ transitions [**?**] and standard ϵ-free NFA-constructions [**?**, **?**, **?**] even $\Omega(m^2)$.)

Coming back to our syntax highlight problem we can use this algorithm to parse the input, build the parse tree from the bit-code and recursively descend it to perform the syntax highlighting. We might (correctly) suspect that the highlighting can be done by piping the bit-code into a separate highlighter process, eliding the materialization of the bit-code.[1] In this paper we show we can do better yet: The algorithm can be generalized to simulating arbitrary nondeterministic finite-state transducers, NFAs with output actions. Furthermore, we can compile their nondeterminism away by producing theoretically and practically very efficient streaming string transducers [**?**, **?**, **?**].

1.1 Contributions

This paper makes the following novel contributions:

- An aggressively *streaming* algorithm for *nondeterministic finite state transducers (NFST)* for ordered output alpabets, which emits the lexicographically least output sequence generated by all accepting paths

[1] All Kleenex code in this paper was highlighted with a Kleenex program emitting LaTeX-commands.

of an input string. It runs in $O(mn)$ time, for automata of size m and inputs of size n.

- An effective determinization of NFSTs into a subclass of *streaming string transducers (SST)* [**?**], finite state machines with string registers that are updated linearly when entering the state upon reading an input symbol. The number of registers required adapts to the number of output actions in the NFST: The fewer output actions the fewer registers. In particular, without special-casing, no registers are generated—yielding a deterministic finite automata (DFA).
- An expressive declarative language, *Kleenex*, for specifying NFSTs with full support for and clear semantics of unrestricted nondeterminism by greedy disambiguation. A basic Kleenex program is a context-free grammar with embedded semantic output actions, but syntactically restricted to ensure that the input is regular.[2] Basic Kleenex programs can be functionally composed into pipelines. The central technical aspect of Kleenex is its semantic support for unbridled (regular) nondeterminism and its effective determinization and compilation to SSTs, thus both highlighting and complementing their significance.
- An implementation, including some empirically evaluated optimizations, of Kleenex that generates SSTs and sequential machines rendered as standard single-threaded C-code, which is eventually compiled to X86 machine code. The optimizations, which are neither conclusive nor final, illustrate the design robustness obtained by the underlying theories of ordered NFST's and SST's.
- Use cases that illustrate the expressive power of Kleenex, and a performance comparison with related tools, including Ragel [**?**], RE2 [**?**] and specialized string processing tools. These document Kleenex's consistently high performance (typically around 1 Gbps, single core, on stock hardware) even when compared to expressively more specialized tools with special-cased algorithms and tools with no or limited support for nondeterminism.

2 Transducers

The semantics of Kleenex will be given by translation to non-deterministic *finite state transducers*, which are finite automata extended with output in a free monoid. In this section, we will recall the standard definition (see e.g. Berstel [**?**]). Since Kleenex is deterministic, we also need to define a disambiguated semantics which allows us to interpret any non-deterministic transducer as a partial function, even when it may have more than one

[2]This facilitates avoiding the $\Omega(M(n))$ lower bound for context-free grammar parsing, where $M(n)$ is the complexity of multiplying $n \times n$ matrices.

possible output for a given input string.

In the following, an *alphabet* is understood to be a finite subset $\{0, 1, ..., n-1\} \subseteq \mathbb{N}$ of consecutive natural numbers with their usual ordering. We fix two alphabets Σ and Γ called the *input* and *output* alphabets, respectively.

Definition 1 (Finite State Transducer)**.** A *finite state transducer* (FST) over Σ, Γ is a structure $\mathcal{T} = (\Sigma, \Gamma, Q, q^-, q^f, E)$ where

- Q is a finite set of *states*;
- $q^-, q^f \in Q$ are the *initial* and *final* states, respectively;
- $E : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \times Q$ is the *transition relation.*

We write $q \xrightarrow{x|y} q'$ whenever $(q, x, y, q') \in E$. The *support* of $q \in Q$ is defined as $\mathrm{supp}(q) = \{x \in \Sigma \cup \{\epsilon\} \mid \exists q', v.\, q \xrightarrow{x|v} q'\}$.

A *path* in $\mathcal{T}$ is a sequence of transitions

$$q_0 \xrightarrow{x_1|y_1} q_1 \xrightarrow{x_2|y_2} ... \xrightarrow{x_n|y_n} q_n$$

It has *input label* $u = x_1x_2...x_n$ and *output label* $v = y_1y_2...y_n$ (ϵ denotes the empty string). We write $q_0 \xrightarrow{u|v} q_n$ if a path from q_0 to q_n with input label u and output label v exists.

$\mathcal{T}$ is *normalized* if for every state $q \in Q_\mathcal{T}$, either $\mathrm{supp}(q) = \{\epsilon\}$ or $\mathrm{supp}(q) \subseteq \Sigma$; and furthermore $\mathrm{supp}(q^f) \subseteq \Sigma$. We write $q\!\downarrow$ for q such that $\mathrm{supp}(q) \subseteq \Sigma$. The formulation of our simulation algorithm in Section 4 becomes simpler when restricting our attention to normalized transducers, since we can take advantage of the following separation property:

Proposition 1. *If $\mathcal{T}$ is normalized, then $p \xrightarrow{uv|z} r\!\downarrow$ if and only if there exists a q such that $z = xy$ and $p \xrightarrow{u|x} q\!\downarrow \xrightarrow{v|y} r\!\downarrow$.*

Definition 2 (Relational Semantics)**.** An FST $\mathcal{T}$ denotes a relation $[\![\mathcal{T}]\!] \subseteq \Sigma^* \times \Gamma^*$ with $(u, v) \in [\![\mathcal{T}]\!]$ iff $q^- \xrightarrow{u|v} q^f$.

The relations definable as FSTs are the *rational relations* [**?**]. In the special case where for any $u \in \Sigma^*$ there is at most one $v \in \Gamma^*$ such that $(u, v) \in [\![\mathcal{T}]\!]$, the transducer computes a partial function. Any FST can be translated to an equivalent normalized FST.

In the following we give a refined semantics which allows us to interpret any FST as denoting a partial function, using the assumed ordering on alphabets to disambiguate between outputs. Our semantics requires restricting paths to be nonproblematic [**?**]: If a path contains a non-empty loop $q' \xrightarrow{\epsilon|v'} q'$ with empty input label, then the path is said to be *problematic*, otherwise it is *nonproblematic*. If there is a nonproblematic path from q to q' with labels u, v, then we write a subscript on the arrow: $q \xrightarrow{u|v}_{\mathsf{np}} q'$.

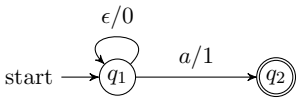
The output words (elements of Γ^*) are lexicographically ordered: $w_1 \leq w_2$ if either w_1 is a prefix of w_2, or there exist words w', w_1', w_2' and symbols $b_1, b_2 \in \Gamma$ such that $w_1 = w'b_1w_1'$, $w_2 = w'b_2w_2'$ and $b_1 < b_2$. We use the ordering on output words to choose a single path from a non-empty set of paths:

Definition 3 (Functional Semantics)**.** Any transducer $\mathcal{T}$ denotes a partial function $[\![\mathcal{T}]\!]_{\leq} : \Sigma^* \to \Gamma^* \cup \{\emptyset\}$ where

$$[\![\mathcal{T}]\!]_{\leq}(u) = \min\{v \mid q^- \xrightarrow{u|v}_{\mathsf{np}} q^f\}.$$

Note that a partial function $A \to B \cup \{\emptyset\}$ is considered here to be a map $A \to 2^B$ where the cardinality of every subset in its range is at most two, and we tacitly identify elements $a \in A$ with their singleton sets $\{a\}$.[3]

Why the restriction to nonproblematic paths? Consider the following transducer $\mathcal{T}$:



Then $\min\{v \mid q_1 \xrightarrow{a|v} q_2\} = \emptyset$, as evidenced by the following infinitely descending chain of outputs: $1 \geq 01 \geq 001 \geq 0001 \geq \ldots$. Operationally, such a chain corresponds to a non-terminating backtracking search. On the other hand, the number of nonproblematic paths with a given input label is always finite, ensuring well-foundedness of the lexicographic order. Every problematic path has a corresponding nonproblematic path with the same input label; consequently, $\mathrm{dom}([\![\mathcal{T}]\!]_{\leq}) = \mathrm{dom}([\![\mathcal{T}]\!])$.

3 Kleenex

The core syntax of Kleenex is essentially that of right regular grammars extended with *output actions* and choice operators. Semantically, a Kleenex program denotes a function which transforms an input string from a regular language into a sequence of action symbols, with the caveat that if the input grammar is ambiguous, then the production rules are chosen according to a greedy leftmost disambiguation strategy.

We will first present the abstract syntax of core Kleenex, which is given a semantics in terms of the transducers introduced in Section 2.

Definition 4 (Kleenex syntax)**.** Let Σ and Γ be two alphabets. A Kleenex program is a non-empty list $p = d_0d_1 \ldots d_n$ of *definitions* d_i, each of the form N_i `:=` t_i, where t_i is a *term* generated by the grammar:

$$t ::= \mathtt{1} \mid N \mid a\ t \mid \mathtt{"}b\mathtt{"}\ t \mid t_0 \,|\, t_1$$

[3] In other words, we adjoin an element to model partial functions as total functions to *pointed sets*.

In the above, N is assumed to range over some set of non-terminal identifiers $\{N_1, ..., N_n\}$, $a \in \Sigma$ over *input symbols* and $b \in \Gamma$ over *output actions*.

We restrict the valid Kleenex programs to those where there is at most one definition for each non-terminal identifier.

Let p be a Kleenex program over non-terminals $\{N_1, \ldots, N_n\}$. We define a set of states Q_p and two transition relations $E_p^{\mathsf{A}}, E_p^{\mathsf{C}}$ as the smallest sets closed under the following rules:

$$\frac{}{N_1 \in Q_p} \qquad \frac{N_i \in Q_p}{t_i \in Q_p \quad (N_i, \epsilon, \epsilon, t_i) \in E_p^{\mathsf{A}} \cap E_p^{\mathsf{C}}} (N_i := t_i)$$

$$\frac{a\ t \in Q_p}{t \in Q_p \quad (a\ t, a, \epsilon, t) \in E_p^{\mathsf{C}} \quad (a\ t, \epsilon, \epsilon, t) \in E_p^{\mathsf{A}}}$$

$$\frac{\texttt{"}b\texttt{"}t \in Q_p}{t \in Q_p \quad (\texttt{"}b\texttt{"}\ t, \epsilon, \epsilon, t) \in E_p^{\mathsf{C}} \quad (\texttt{"}b\texttt{"}\ t, \epsilon, b, t) \in E_p^{\mathsf{A}}}$$

$$\frac{t_0 \,|\, t_1 \in Q_p}{\{t_0, t_1\} \subseteq Q_p \quad (t_0 \,|\, t_1, \epsilon, 0, t_0), (t_0 \,|\, t_1, \epsilon, 1, t_1) \in E_p^{\mathsf{C}} \quad (t_0 \,|\, t_1, 0, \epsilon, t_0), (t_0 \,|\, t_1, 1, \epsilon, t_1) \in E_p^{\mathsf{A}}}$$

The sets are easily seen to be finite. They define two transducers, an *oracle* $\mathcal{T}_p^{\mathsf{C}} = (\Sigma, \mathbb{2}, Q_p, N_1, \mathtt{1}, E_p^{\mathsf{C}})$ and an *action machine* $\mathcal{T}_p^{\mathsf{A}} = (\mathbb{2}, \Gamma, Q_p, N_1, \mathtt{1}, E_p^{\mathsf{A}})$, where $\mathcal{T}_p^{\mathsf{A}}$ is easily seen to be deterministic, and $\mathcal{T}_p^{\mathsf{C}}$ is non-deterministic and possibly ambiguous. The oracle intuitively translates an input string to a set of codes for the possible paths through p which reads the given string. The action machine translates a code to a sequence of actions.

Disambiguating according to the greedy leftmost strategy corresponds to choosing the lexicographically least code, and we can thus define the semantics as follows:

Definition 5 (Kleenex semantics)**.** Let p be a Kleenex program and let $\mathcal{T}_p^{\mathsf{C}}$ and $\mathcal{T}_p^{\mathsf{A}}$ be defined as above. The program p denotes a partial function $[\![p]\!] : \Sigma^* \to \Gamma^* \cup \{\emptyset\}$ given by

$$[\![p]\!] = [\![\mathcal{T}_p^{\mathsf{A}}]\!] \circ [\![\mathcal{T}_p^{\mathsf{C}}]\!]_{\leq}$$

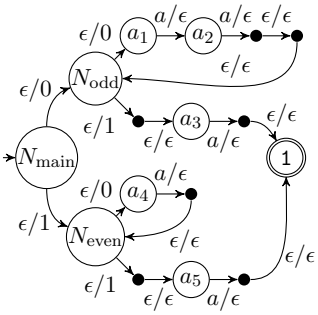
3.1 Syntactic sugar

The full syntax of our language is obtained by extending the syntax of core Kleenex with the following term-level constructors:

$$\begin{aligned} t ::= \ldots \mid \texttt{"}v\texttt{"} \mid \texttt{/}e\texttt{/} \mid \texttt{\~{}}t \mid t_0 \cdot t_1 \mid t\texttt{*} \mid t\texttt{+} \mid t\texttt{?} \\ \mid t\{n\} \mid t\{n\texttt{,}\} \mid t\{\texttt{,}m\} \mid t\{n\texttt{,}m\} \end{aligned}$$

```
main  := odd  ~/a/
       | even ~/a/
odd   := ~/aa/ "bb" odd
       | "c"
even  := ~/a/ "c" even
       | "b"
```

$$
\begin{aligned}
N_{\text{main}} &:= N_{\text{odd}} \mid N_{\text{even}} \\
N_{\text{odd}} &:= a\ a\ \texttt{"}b\texttt{"}\ \texttt{"}b\texttt{"}\ N_{\text{odd}} \\
&\quad \mid \texttt{"}\ c\ \texttt{"}\ a\ 1 \\
N_{\text{even}} &:= a\ \texttt{"}c\texttt{"}\ N_{\text{even}} \mid \texttt{"}b\texttt{"}\ a\ 1
\end{aligned}
$$



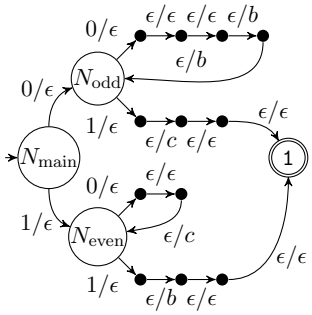


Figure 1: In the top left is a Kleenex program in the surface syntax and on the right is the desugared version. Below, the oracle transducer and action machine is shown, from left to right. The transduction realized by the program maps a^{2n+1} to $b^{2n}c$, and a^{2n+2} to $c^{2n}b$, respectively.

where $v \in \Gamma^*$, $n, m \in \mathbb{N}$, and e is a *regular expression*. The term `"`v`"` is just shorthand for a sequence of action symbols. The regular expressions are special versions of Kleenex terms that do not contain identifiers. They always desugar to terms that output the matched input string: The sugared term `/`e`/` adds a default action `"`$\alpha(a)$`"` after every input symbol a in e using a given default action map $\alpha : \Sigma \to \Gamma$. For example, the regular expression `/a*|b{n,m}|c?/` becomes the term $(a\texttt{"}a\texttt{"})\texttt{*}\mid(b\texttt{"}b\texttt{"})\{n\texttt{,}m\}\mid(c\texttt{"}c\texttt{"})\texttt{?}$. A *suppressed* subterm is written `~`t, and it desugars into t with all action symbols removed. Composition $t_0 \cdot t_1$ allows general sequential composition instead of the strict cons-like form of the core syntax. The operators $\cdot$`*`, $\cdot$`+` and $\cdot$`?` desugar to their usual meaning as regular operators, as do operators $\cdot\{n\}$, $\cdot\{n\texttt{,}\}$, $\cdot\{\texttt{,}m\}$, and $\cdot\{n\texttt{,}m\}$.

By convention, the nonterminal named `main` is the entry point to a Kleenex program.

The desugaring can be described more precisely by a desugaring operator $\mathcal{D}[\![\cdot, \cdot]\!]$. The result of desugaring a program $p = d_1 \ldots d_n$ with initial term N_1`:=` t_1 is a program with initial term N_1'`:=` $\mathcal{D}[\![t_1, 1]\!]$ which furthermore is

a solution to the following set of equations:

$$
\begin{aligned}
\mathcal{D}[\![\mathtt{1}, k]\!] &= \mathcal{D}[\![\mathtt{\sim 1}, k]\!] = k \\
\mathcal{D}[\![\mathtt{"}b_1 \ldots b_n\mathtt{"}, k]\!] &= \mathtt{"}b_1\mathtt{"} \ldots \mathtt{"}b_n\mathtt{"}\ k \\
\mathcal{D}[\![\mathtt{\sim("}b\mathtt{"}\ t), k]\!] &= \mathcal{D}[\![\mathtt{\sim}t, k]\!] \\
\mathcal{D}[\![a\ t, k]\!] &= a\ \mathcal{D}[\![t, k]\!] \\
\mathcal{D}[\![\mathtt{\sim}(a\ t), k]\!] &= a\ \mathcal{D}[\![\mathtt{\sim}t, k]\!] \\
\mathcal{D}[\![\mathtt{\sim}(t_0 \mathtt{|}\, t_1), k]\!] &= \mathcal{D}[\![\mathtt{\sim}t_0, k]\!] \mathtt{|}\, \mathcal{D}[\![\mathtt{\sim}t_1, k]\!] \\
\mathcal{D}[\![N, k]\!] &= N_{\mathcal{D}[\![t,k]\!]} && \text{(where } N \mathtt{:=} t\text{)} \\
\mathcal{D}[\![\mathtt{\sim}N, k]\!] &= N_{\mathcal{D}[\![\mathtt{\sim}t,k]\!]} && \text{(where } N \mathtt{:=} t\text{)} \\
\mathcal{D}[\![\mathtt{/}e\mathtt{/}, k]\!] &= \mathcal{D}[\![t_e, k]\!] \\
\mathcal{D}[\![t_0 \cdot t_1, k]\!] &= \mathcal{D}[\![t_0, \mathcal{D}[\![t_1, k]\!]]\!] \\
\mathcal{D}[\![t_0 \mathtt{|}\, t_1, k]\!] &= \mathcal{D}[\![t_0, k]\!] \mathtt{|}\, \mathcal{D}[\![t_1, k]\!] \\
\mathcal{D}[\![t\mathtt{*}, k]\!] &= \mathcal{D}[\![t, \mathcal{D}[\![t\mathtt{*}, k]\!]]\!] \mathtt{|}\, k \\
\mathcal{D}[\![t\mathtt{+}, k]\!] &= \mathcal{D}[\![t, \mathcal{D}[\![t\mathtt{*}, k]\!]]\!] \\
\mathcal{D}[\![t\mathtt{?}, k]\!] &= \mathcal{D}[\![t, k]\!] \mathtt{|}\, k
\end{aligned}
$$

In the above, a non-terminal name N_t on the right-hand side of an equation implies the presence of a definition $N_t \mathtt{:=} t$, and the term t_e corresponds to the regular expression e as described above. The range patterns are just expanded and then further desugared.

The system does not always have a well-defined solution: The generalized composition operator of sugared Kleenex allows one to write non-regular grammars, for example:

$$A \mathtt{:=} (aA) \cdot b \mathtt{|} \mathtt{1}.$$

A program that does not have a well-defined desugaring is not considered to be well-formed, and we will not attempt to give it a semantics.

3.2 Custom register updates

We extend the syntax of Kleenex further with *register actions*:

$$
\begin{aligned}
t ::= \ldots \ &| \ R \mathtt{\ @\ } t \ | \ \mathtt{!}R \\
&| \ \mathtt{[}\ R \mathtt{\ <-\ } (R \mid \mathtt{"}v\mathtt{"})^\star \mathtt{\ ]} \\
&| \ \mathtt{[}\ R \mathtt{\ +=\ } (R \mid \mathtt{"}v\mathtt{"})^\star \mathtt{\ ]}
\end{aligned}
$$

where R is a lower-case register name. Intuitively, these constructs allow one to store actions and perform them later. Writing $R \mathtt{\ @\ } t$ redirects all actions that would have resulted from running t into the register R, which can be performed later by writing $\mathtt{!}R$. The register R can be either set to a

sequence of actions $(R \mid \texttt{"}v\texttt{"})^\star$ or appended with them, using the `<-` and `+=` construct, respectively.

At first glance it seems like adding custom register updates to Kleenex significantly alters the language and moves beyond the semantics discussed so far. However, the only requirement on the output actions is that they form a monoid, so this is not the case. We simply add actions like "set register R to v" as output symbols along with the output symbols $\texttt{!}v$. The output redirection caused by the $\cdot$ `@` $\cdot$ operator can be understood as a push operation: when R `@` t is written it means that in the scope of t the topmost register is R. If there are other redirection symbols in t, these will come in and out of scope as they are pushed and popped to the stack.

As an example, the following program swaps two input lines by storing them in registers `a` and `b` and outputting them in reverse order:

```
main := a@line b@line !b !a
line := /[^\n]*\n/
```

4 Simulation and determinization

In this section, we specify an algorithm for simulating FSTs under the functional semantics, allowing us to efficiently evaluate the oracle transducer defined in Section 3. We also show how the simulation algorithm can be implemented by finite deterministic *streaming string transducers* [**?**] whose states are identified by equivalence classes of simulation states. The latter construction gives a deterministic machine model for Kleenex programs which can be compiled to efficient code for executing on hardware.

We note that non-deterministic transducers are strictly more powerful than their deterministic counterparts, and can thus not always be determinized in general. Determinization procedures exist [**?**, **?**] which result in a deterministic transducer with an infinite state set in the general case, and a finite state set if and only if the underlying transduction is *subsequential* [**?**, **?**]. The oracle transducers of Kleenex programs are not subsequential in general. Our simulation algorithm is also different from the existing methods for determinizing transducers by also taking disambiguation into account.

In the following we fix a transducer $\mathcal{T} = (\Sigma, \Delta, Q, q^-, q^f, E)$. We will assume that $\mathcal{T}$ is normalized, and that it furthermore satisfies the following property:

Definition 6 (Prefix-free transducer)**.** $\mathcal{T}$ is said to be *prefix-free* if for all $p, q, q' \in Q_\mathcal{T}$ where $\text{supp}(q), \text{supp}(q') \subseteq \Sigma$ we have that if $p \xrightarrow{x|y} q$ and $p \xrightarrow{x|y'} q'$ then $y \not\asymp y'$.

It is easy to verify that the oracle transducers constructed Section 3 are

both normalized and prefix-free. Note that they will always have $\Delta = \mathbb{2}$, but our construction generalizes to oracle alphabets of all sizes.

4.1 Generalized state set simulation

Let D be a finite and totally ordered set, and write $S(D, Q)$ for the set of partial functions $Q \to D^* \cup \{\emptyset\}$. Elements $A \in S(D, Q)$ can be seen as generalized subsets of Q where every member q is labeled by some element $A(q) \in D^*$, and every non-member has $A(q) = \emptyset$.

We extend word concatenation in D^* to the set $D^* \cup \{\emptyset\}$ by setting $x\emptyset = \emptyset = \emptyset x$. For $u, v \in D^* \cup \{\emptyset\}$, write $u \preceq v$ if u is a *prefix* of v, i.e. there is a unique w such that $v = uw$. Write $u \prec v$ if w has length at least one. Let $u \wedge v$ refer to the longest p such that $u = pu'$ and $v = pv'$ for some u', v'. Note that in view of this definition, $\emptyset$ becomes a neutral element with $u \wedge \emptyset = u = \emptyset \wedge u$.

We define a right action $\cdot : S(D, Q) \times \Sigma^* \to S(D \cup \Delta, Q)$ on the generalized state sets as follows:

Definition 7 (Right action)**.** Let $A \in S(D, Q)$ and $u \in \Sigma^*$. We define

$$(A \cdot u)(q) = \min\{A(p)v \mid p \xrightarrow{u|v}_{\mathsf{np}} q \downarrow\}.$$

When $D = \Delta$ the right action can be seen as a map $S(\Delta, Q) \times \Sigma^* \to S(\Delta, Q)$. It is easily seen that the right action is related to the functional semantics in the following way:

Proposition 2. *Let $A(q) = \epsilon$ if $q = q^-$ and $A(q) = \emptyset$ otherwise. Then $(A \cdot u)(q^f) = [\![\mathcal{T}]\!]_{\leq}(u)$.*

A generalized subset $A \in S(D, Q)$ is said to be *prefix-free* if $A(p) \not\prec A(q)$ for all $p, q \in Q$. When $\mathcal{T}$ is normalized and prefix-free, the right action preserves prefix-freeness of generalized subsets and commutes with word concatenation:

Proposition 3. *If $\mathcal{T}$ is normalized and prefix-free and A is prefix-free, then for all $u, v \in \Sigma^*$,*

1. *$A \cdot u$ is prefix-free; and*
2. *$(A \cdot u) \cdot v = A \cdot uv$.*

Proof. The first property follows directly by A and $\mathcal{T}$ being prefix-free. For

the second, we have for $r \in Q$,

$$\begin{aligned}
&((A \cdot u) \cdot v)(r) \\
&= \min\{(A \cdot u)(q)y \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
&= \min\{\min\{A(p)x \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow\}y \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
&= \min\{\min\{A(p)xy \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow\} \mid q \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
&= \min\{A(p)xy \mid p \xrightarrow{u|x}_{\mathsf{np}} q \downarrow \xrightarrow{v|y}_{\mathsf{np}} r \downarrow\} \\
&= \min\{A(p)z \mid p \xrightarrow{uv|z}_{\mathsf{np}} r \downarrow\} = (A \cdot uv)(r)
\end{aligned}$$

The third equality is a consequence of the fact that A and $\mathcal{T}$ are prefix-free, together with the following easily proved fact about lexicographic ordering: $(\min X)y = \min\{xy \mid x \in X\}$ whenever X is a set of pairwise prefix-free words. The fourth equality is just associativity of minimum, and the last equality follows by the fact that $\mathcal{T}$ is normalized and Proposition 1. □

For $x \in D^*$ and $A \in S(D, Q)$, define $xA \in S(D, Q)$ by $(xA)(q) = x(A(q))$. We say that x is a prefix of A if $A = xA'$ for some A', which is equivalent to x being a prefix of every $A(q)$. The right action commutes with the prefix operation:

Proposition 4. *Let $x \in D^*$, then $(xA) \cdot u = x(A \cdot u)$ for all $u \in \Sigma^*$.*

Proof. Follows by the fact that lexicographic ordering satisfies $\min\{xy \mid y \in Y\} = x \min Y$. □

Streaming simulation algorithm A streaming simulation algorithm on $\mathcal{T}$ processes an input from left to right and may write zero or more symbols to the output in each step.

Algorithm 1 (Streaming FST Simulation)**.** Let $\mathcal{T}$ be a normalized and prefix-free transducer, and let the input $u = a_1 a_2 ... a_n$ be given. Let $A_0 \in S(\Delta, Q)$ be defined as in Proposition 2. Reading symbol a_i, compute $B_i = A_i \cdot a_{i+1}$. Append $p_i = \bigwedge_{q \in Q} B_i(q)$ to the output stream and set $A_{i+1} = B_i'$, where the equality $B_i = p_i B_i'$ defines B_i'. When there are no more input symbols left, append $(A_n \cdot \epsilon)(q^f)$ to the output and return, or fail if $(A_n \cdot \epsilon)(q^f) = \emptyset$.

By Proposition 2, Proposition 3 and Proposition 4, the algorithm computes $[\![\mathcal{T}]\!]_{\leq}(u)$.

4.2 A deterministic computation model

We wish to translate Kleenex programs to completely deterministic programs without a simulation overhead.

Single-valued transducers in general, however, can only be determinized if the underlying function is *subsequential* [**?**, **?**], a property which is not satisfied in general for the oracle transducers constructed from Kleenex programs.

We turn instead to *streaming string transducers* [**?**] (SST), a deterministic model of computation which generalizes subsequential transducers by allowing copy-free updates to a finite set of word registers. It turns out that every transducer that can be simulated by our generalized state set algorithm can be expressed as an SST.

Definition 8 (Streaming String Transducer [**?**])**.** A deterministic *streaming string transducer* (SST) over alphabets Σ, Δ is a structure $\mathcal{S} = (X, Q, q^-, F, \delta^1, \delta^2)$ where

- X is a finite set of *register variables*;
- Q is is a finite set of *states*;
- F is a partial function $Q \to (\Delta \cup X)^* \cup \{\emptyset\}$ mapping each *final state* $q \in \mathrm{dom}(F)$ to a word $F(q) \in (\Delta \cup X)^*$ such that each $x \in X$ occurs at most once in $F(q)$;
- δ^1 is a transition function $Q \times \Sigma \to Q$;
- δ^2 is a *register update* function $Q \times \Sigma \times X \to (\Delta \cup X)^*$ such that for each $q \in Q$, $a \in \Sigma$ and $x \in X$, there is at most one $y \in X$ such that x occurs in $\delta^2(q, a, y)$.

The semantics are defined as follows. A *configuration* of an SST $\mathcal{S}$ is a pair (q, ρ) where $q \in Q_{\mathcal{S}}$ is a state, and $\rho : X_{\mathcal{S}} \to \Delta^*$ is a *valuation*. A valuation extends as a monoid homomorphism to a map $\widehat{\rho} : (X_{\mathcal{S}} \cup \Delta)^* \to \Delta^*$ by setting $\rho(x) = x$ for $x \in \Delta$. The initial configuration is (q^-, ρ^-) where $\rho^-(x) = \epsilon$ for all $x \in X_{\mathcal{S}}$.

A configuration steps to a new configuration given an input symbol: $\delta_{\mathcal{S}}((q, \rho), a) = (\delta^1_{\mathcal{S}}(q, a), \rho')$, where $\rho'(x) = \widehat{\rho}(\delta^2_{\mathcal{S}}(q, a, x))$. The transition function extends to a transition function on words $\delta^*_{\mathcal{S}}$ by $\delta^*_{\mathcal{S}}((q, \rho), \epsilon) = (q, \rho)$ and $\delta^*_{\mathcal{S}}((q, \rho), au) = \delta^*_{\mathcal{S}}(\delta_{\mathcal{S}}((q, \rho), a), u)$.

Every SST $\mathcal{S}$ denotes a partial function $[\![\mathcal{S}]\!] : \Sigma^* \to \Delta^* \cup \{\emptyset\}$ where for any $u \in \Sigma^*$, we define

$$[\![\mathcal{S}]\!](u) = \begin{cases} \widehat{\rho'}(F_{\mathcal{S}}(q')) & \text{if } \delta^*((q^-, \rho^-), u) = (q', \rho') \\ & \text{and } q' \in \mathrm{dom}(F_{\mathcal{S}}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

4.3 Tabulation

We need to come up with a representation of our streaming simulation algorithm as an SST with a designated register used for streaming output. Our representation needs to satisfy the property of being finite state as well

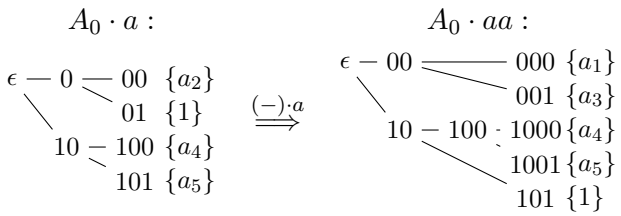
as the property that the output register contains the output $p_1 p_2 ... p_i$ of Algorithm 1 after reading input symbol a_i. The latter requirement means that we must somehow statically encode the prefix structure of all potential outputs in the states of the SST, since SSTs cannot access the contents of registers. It turns out that this is possible by letting the states of the SST be equivalence classes of generalized state sets, where the equivalence relates state sets that agree on state ordering and prefix structure.

Trees We will call a prefix-free generalized state set A an *ordered tree* with node set

$$N_A = \{A(p) \wedge A(q) \mid p, q \in Q, A(p) \wedge A(q) \neq \emptyset\}.$$

Under this view, the leaves of A seen as a tree is the subset of nodes $L_A = \{A(q) \mid q \in Q, A(q) \neq \emptyset\} \subseteq N_A$, and the leaves are labeled by $A^{-1} : L_A \to 2^Q$. Since A is assumed prefix-free, we have for any nodes $x, y \in N_A$ that $x \preceq y$ if and only if there is a $z \in N_A$ such that $x = y \wedge z$. In this case x is called an *ancestor* of y and z, which in turn are called the *descendants* of x. Importantly, the root node of any (sub)tree is the longest common prefix of its descendants.

Example 1. We illustrate the tree interpretation as follows. Consider the oracle transducer from Figure 1. Let A_0 be the generalized state set that maps N_{main} to ϵ and every other state to $\emptyset$. Then the state sets $A_0 \cdot a$ and $A_0 \cdot aa$ can be seen as trees in the following way:



We will consider two generalized state sets to be equivalent if they are indistinguishable as ordered trees.

Definition 9 (Ordered tree isomorphism)**.** Let D_1, D_2 be totally ordered and let $A_1 \in S(D_1, Q)$ and $A_2 \in S(D_2, Q)$ be trees. An ordered tree isomorphism between A_1 and A_2 is a bijective map $h : N_{A_1} \to N_{A_2}$ such that for all $p, q \in Q$:

1. $h(A_1(p) \wedge A_1(q)) = A_2(p) \wedge A_2(q)$; and
2. $A_1(p) \leq A_1(q)$ if and only if $A_2(p) \leq A_2(q)$.

We write $h : A_1 \equiv A_2$ and say that A_1 and A_2 are *equivalent* when h is an ordered tree isomorphism between A_1 and A_2. Tree equivalence is preserved by the right action:

Proposition 5. *If $A \in (D_1, Q), B \in (D_2, Q)$ and $h : A \equiv B$ then for all $a \in \Sigma$, we have $A \cdot a \equiv B \cdot a$.*

Proof sketch. Since h is an order isomorphism and since A and B are prefix-free, we have for all $q \in Q$ exists $p_q \in Q$ and $y_q \in \Delta^*$ such that $(A \cdot a)(q) = A(p_q)y_q$ and $(B \cdot a)(q) = h(A(p_q))y_q$. Observe that for any $n \in N_{A \cdot a}$ there exists $q_1, q_2 \in Q$ such that

$$\begin{aligned} n &= (A \cdot a)(q_1) \wedge (A \cdot a)(q_2) \\ &= \begin{cases} A(q_1)(y_{q_1} \wedge y_{q_2}) & \text{if } A(q_1) = A(q_2) \\ A(q_1) \wedge A(q_2) & \text{otherwise} \end{cases} \end{aligned}$$

Furthermore, there does not exist $q_1, q_2, r_1, r_2 \in Q$ such that $A(q_1)(y_{q_1} \wedge y_{q_2}) = A(r_1) \wedge A(r_2)$, since that would imply that $A(q_1)$ is a prefix of $A(r_1)$ and $A(r_2)$. We define a map $h' : N_{A \cdot a} \to N_{B \cdot a}$ such that for all $q_1, q_2 \in Q$,

$$\begin{aligned} &h'((A \cdot a)(q_1) \wedge (A \cdot a)(q_2)) \\ &\quad = \begin{cases} h(A(q_1)(y_{q_1} \wedge y_{q_2})) & \text{if } A(q_1) = A(q_2) \\ h(A(q_1) \wedge A(q_2)) & \text{otherwise.} \end{cases} \end{aligned}$$

This is a well-defined function by the previous observations, and a tree isomorphism by the fact that h is a tree isomorphism. □

Canonical representatives Call a generalized set $A \in S(D, Q)$ *canonical* if

1. $\mathrm{rng}(A)$ is prefix closed: if $y \in \mathrm{rng}(A)$ and $x \preceq y$ then $x \in \mathrm{rng}(A)$; and
2. $\mathrm{rng}(A)$ is downwards closed: if $x\ b \in \mathrm{rng}(A)$ for $b' < b$ then $xb' \in \mathrm{rng}(A)$ (for $b, b' \in \Delta$).

Write $\widetilde{S}(D, Q)$ for the subset of canonical trees. The set is finite, as every canonical tree A has a prefix closed node set, so the longest word in N_A is bounded by $|\mathrm{dom}(A)| - 1$ (the maximum depth of a tree with $|\mathrm{dom}(A)|$ leaves).

Any tree has a canonical representative:

Proposition 6. *For any set D and tree $A \in S(D, Q)$, there is a unique $C \in \widetilde{S}(\mathbb{N}, Q)$ with $A \equiv C$.*

As a consequence, there is a reduction map $[\cdot] : S(D, Q) \to \widetilde{S}(\mathbb{N}, Q)$ such that $A \equiv B$ if and only if $[A] = [B]$, implying that the quotient set $S(D, Q)/\equiv$ must be finite. Any $A \in S(D, Q)$ is thus canonically represented by a homomorphism $h_A : N_{[A]} \to N_A$ such that $A = h_A \circ [A]$.

In view of Proposition 5, this means that we can statically enumerate all possible trees up to tree isomorphism by computing with the canonical

representatives. Any concrete tree reachable by the simulation algorithm is an instance of a canonical tree composed with a suitable homomorphism. An SST implementing the simulation algorithm can thus take the set of canonical trees as its states, and will then need to maintain the associated homomorphism via register updates.

Paths We need to represent tree homomorphisms using SST registers such that the effect of computing right actions on the underlying tree can be expressed as SST updates.

For a tree $A \in S(D,Q)$, any node $x \in N_A$ has a unique maximal decomposition $x = x_0x_1...x_n$ such that each $x_0x_1...x_i \in N_A$ for all $0 \leq i \leq n$. Intuitively, this reflects the full path from the root node to the node x, and we can define the map

$$\begin{aligned} \mathsf{path}_A &: N_A \to N_A^* \\ \mathsf{path}_A(x) &= (x_0, x_0x_1, ..., x_0x_1...x_n), \end{aligned}$$

which maps nodes to their maximal path decomposition (we use the tuple notation to distinguish between the two levels of monoids). In view of this and the fact that homomorphisms must preserve descendants, then for any homomorphism $h : A \equiv B$ there is a unique $\kappa_h : N_A \to N_B$ such that

$$h(x) = \kappa_h(t_0)\kappa_h(t_1)\cdots\kappa_h(t_n), \tag{1}$$

where $\mathsf{path}_A(x) = (t_0, t_1, ..., t_n)$. Intuitively, κ can be seen as a "differential" representation of h, representing the change of h between a node and its immediate ancestor. By viewing κ_h as a map $N_A \to D_B^*$ which extends uniquely to a monoid homomorphism $\widehat{\kappa_h} : N_A^* \to D_B^*$, we obtain $h = \widehat{\kappa_h} \circ \mathsf{path}_A$. Considering the unique isomorphism $h_A : [A] \equiv A$, write κ_A for the associated decomposition satisfying (1), and we thus have

$$A = \widehat{\kappa_A} \circ \mathsf{path}_{[A]} \circ [A] \tag{2}$$

The path-operator is easily seen to be a tree isomorphism since it preserves node ordering and prefix structure. That is, for any $A \in S(D,Q)$, we have $\mathsf{path}_A : A \equiv A^\sharp$ where $A^\sharp \in S(N_D, Q)$ is defined by $A^\sharp = \mathsf{path}_A \circ A$. Using this notation, (2) becomes

$$A = \widehat{\kappa_A} \circ [A]^\sharp, \tag{3}$$

SST construction We construct an SST implementing the FST simulation algorithm and sketch a proof of its correctness.

Theorem 1. *For any normalized prefix-free transducer $\mathcal{T} = (\Sigma, \Delta, Q, q^-, q^f, E)$, there is an SST $\mathcal{S}$ such that $[\![S]\!] = [\![T]\!]_\leq$.*

Proof. We define $\mathcal{S}$ as follows. Let A_0 be defined as in Algorithm 1, and observe that $A_0 \in \widetilde{S}(\mathbb{N}, Q)$. The states are the canonical trees labeled by Q:

$$\begin{aligned} Q_{\mathcal{S}} &= \{[A] \mid A \in S(\Delta, Q)\} \cup \{A_0\} \subseteq \widetilde{S}(\mathbb{N}, Q), \\ q^-_{\mathcal{S}}(q) &= A_0(q) \end{aligned}$$

The registers will be identified by canonical tree nodes:

$$X_{\mathcal{S}} = \bigcup \{N_C \mid C \in Q_{\mathcal{S}}\}.$$

The final output and the transition maps are given as follows:

$$\begin{aligned} F_{\mathcal{S}}(C) &= (C^\sharp \cdot \epsilon)(q^f), \\ \delta^1_{\mathcal{S}}(C, a) &= [C \cdot a], \\ \delta^2_{\mathcal{S}}(C, a, x) &= \begin{cases} \kappa_{C^\sharp \cdot a}(x) & \text{if } x \in N_{[C^\sharp \cdot a]} \\ \epsilon & \text{otherwise} \end{cases} \end{aligned}$$

We claim that $\mathcal{S}$ computes the same function as $\mathcal{T}$ under the functional semantics.

For $u \in \Sigma^*$ let (C_u, ρ_u) refer to the value $\delta^*_{\mathcal{S}}((q^-_{\mathcal{S}}, \rho^-), u) = (C_i, \rho_i)$. We show that for any $u \in \Sigma^*$, we have $\widehat{\rho_u} \circ (C^\sharp_u \cdot \epsilon) = A_0 \cdot u$.

Suppose that this holds. Then for any $u \in \Sigma^*$, we have by the above and Proposition 2 that $[\![S]\!](u) = \widehat{\rho_u}(F_{\mathcal{S}}(C_u)) = \widehat{\rho_u} \circ (C^\sharp_u \cdot \epsilon)(q^f) = (A_0 \cdot u)(q^f) = [\![\mathcal{T}]\!]_{\leq}(u)$.

Our claim follows as a special case of the following lemma. □

Lemma 1. *Let $A \in S(\Delta, Q)$ and $\rho : X_{\mathcal{S}} \to \Delta^*$ such that $A = \widehat{\rho} \circ [A]^\sharp$. Then for any $u \in \Sigma^+$ with $\delta^*_{\mathcal{S}}(([A], \rho), u) = (C, \rho')$ we have $\widehat{\rho'} \circ C^\sharp = A \cdot u$.*

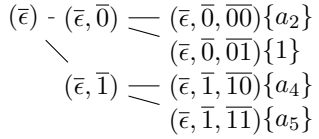
Proof. By induction on u. For $u = a$ we have $C = [[A] \cdot a] = [A \cdot a]$ and $\rho' = \widehat{\rho} \circ \kappa_{[A]^\sharp \cdot a}$. We can easily verify that $\widehat{\rho'} = \widehat{\rho} \circ \widehat{\kappa_{[A]^\sharp \cdot a}}$ so for any $q \in Q$,

$$\begin{aligned} \widehat{\rho'} \circ [A \cdot a]^\sharp(q) &= \widehat{\rho} \circ \widehat{\kappa_{[A]^\sharp \cdot a}} \circ [A \cdot a]^\sharp(q) \\ &= \widehat{\rho} \circ ([A]^\sharp \cdot a)(q) \\ &= \widehat{\rho}(\min\{[A]^\sharp(p)y \mid p \xrightarrow{a|y}_{\mathsf{np}} q \downarrow\}) \\ &= \min\{\widehat{\rho} \circ [A]^\sharp(p)y \mid p \xrightarrow{a|y}_{\mathsf{np}} q \downarrow\} \\ &= \min\{A(p)y \mid p \xrightarrow{a|y}_{\mathsf{np}} q \downarrow\} = (A \cdot a)(q) \end{aligned}$$

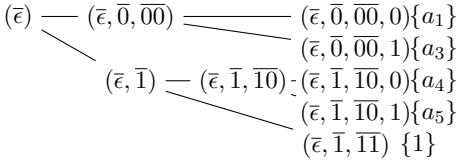
The second equality follows by observing that $A \equiv [A] \equiv [A]^\sharp$, so by Proposition 5, we have $A \cdot a \equiv [A]^\sharp \cdot a$ and thus $[A \cdot a] = [[A]^\sharp \cdot a]$. Therefore, $\widehat{\kappa_{[A]^\sharp \cdot a}} \circ [A \cdot a]^\sharp = \widehat{\kappa_{[A]^\sharp \cdot a}} \circ [[A]^\sharp \cdot a]^\sharp = [A]^\sharp \cdot a$ by using the identity (3). The fourth equality is justified by the fact that $[A]^\sharp(p) \leq [A]^\sharp(q)$ if and only if $A(p) \leq A(q)$.

For $u = au'$ where $u' \neq \epsilon$, we have $(C, \rho') = \delta^*_{\mathcal{S}}(([A \cdot a], \widehat{\rho} \circ \kappa_{[A]^\sharp \cdot a})$. By the previous argument we can apply the induction hypothesis, and we obtain $C = [(A \cdot a) \cdot u']$ and $\widehat{\rho'} \circ C^\sharp = (A \cdot a) \cdot u'$. The result then follows by Proposition 3. □

Example 2. We illustrate how the construction works by showing how Example 1 is implemented as an SST update between states $[A_0 \cdot a]$ and $[A \cdot aa]$. The register update is obtained by computing $\kappa_{[A_0 \cdot a]^\sharp \cdot a}$. The tree $[A_0 \cdot a]^\sharp$ looks as follows:



Recall that each node is a full path in the canonical tree $[A_0 \cdot a]$. The node names from $N_{[A_0 \cdot a]}$ are overlined and elements of the path monoid $N^*_{[A_0 \cdot a]}$ is written $(x_1, x_2, ...)$. The tree $[A_0 \cdot a]^\sharp \cdot a$ looks as follows:



Note that symbols that are not overlined are output symbols from Δ. The map $\kappa' = \kappa_{[A_0 \cdot a]^\sharp \cdot a} : N_{[A_0 \cdot aa]} \to (N_{[A_0 \cdot a]} \cup \Delta)^*$ gives us the relevant SST update strings:

$$\begin{array}{lll} \kappa'(\overline{\epsilon}) = (\overline{\epsilon}) & \kappa'(\overline{0}) = (\overline{0}, \overline{00}) & \kappa'(\overline{00}) = 0 \\ \kappa'(\overline{01}) = 1 & \kappa'(\overline{1}) = (\overline{1}) & \kappa'(\overline{10}) = (\overline{10}) \\ \kappa'(\overline{100}) = 0 & \kappa'(\overline{101}) = 1 & \kappa'(\overline{11}) = (\overline{11}) \end{array}$$

The full construction of an SST from the oracle transducer in Figure 1 can be seen in Figure 2.

5 Implementation

Our implementation compiles a Kleenex program to machine code by implementing the transducer constructions described in the earlier sections. We have also implemented several optimizations to decrease the size of the generated SSTs and improve the performance of the generated code. We will briefly describe these in the following section, and we note that they are all orthogonal to the underlying principles behind our compilation.

The possible compilation paths of our implementation can be seen in Fig. 3.

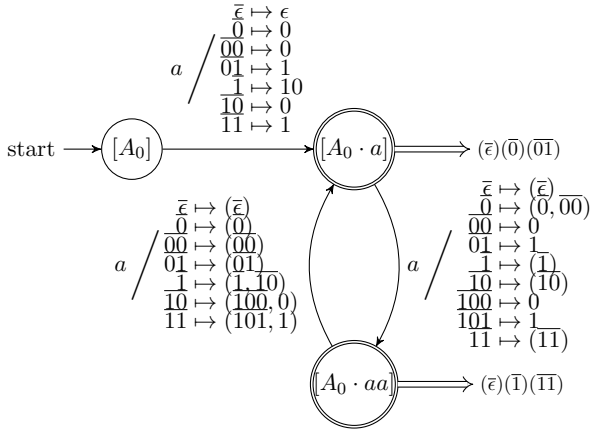


Figure 2: Example of SST computing the same function as the oracle transducer in Figure 1. Each transition is tagged by a register update, and the nodes of the canonical tree identifying the destination state make up the registers. The wide arrows exiting the accepting states indicate the final output string. Note that this always includes the root variable ($\overline{\epsilon}$) which thus acts as an interface for streaming output (although for this particular example, nothing can output until the end of the input).

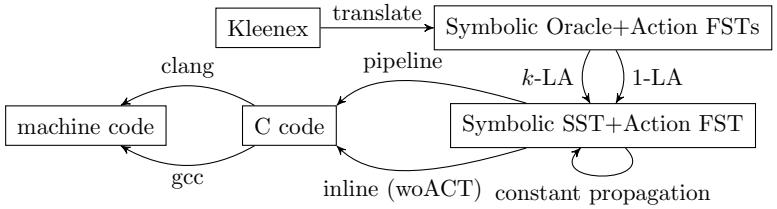


Figure 3: Compilation paths. 1-LA is symbolic SST construction with single-symbol transitions; k-LA is construction of SST with up to k symbols of lookahead for some k determined by the program. The "pipeline" translation path indicates that the resulting program keeps the oracle SST and action FST separate, with data being piped from the SST to the FST at runtime. The "inline" path indicates that the action FST is fused into the oracle SST.

5.1 Transducer pipeline

It is possible to chain together several Kleenex programs in a pipeline, letting the output of one serve as the input of the next. This can for example be used to strip unwanted characters before performing a transformation. By using the optional *pipeline pragma*, `start:` t_1 `>>` ... `>>` t_n, a programmer can specify that the entry point is t_1 and that the output should be chained together as specified, with the final output being that of t_n. The implementation does this by spawning a process for each transducer and setting up UNIX pipes between them.

5.2 Inlining the action transducer

When we have constructed the oracle SST we end up with two deterministic machines which need to be composed. We can either do this at runtime, piping the output of the oracle SST into the action FST, or we can apply a form of deforestation to inline the outuput of the action FST directly in the SST (this is straightforward since the action FST is deterministic by construction). The former approach is advantageous if the Kleenex program produces a lot of output and is highly nondeterministic.

5.3 Constant propagation

The SSTs generated by our construction contains quite a lot of trivial register updates which can be eliminated in order to achieve better run-time efficiency. Consider the SST in Fig. 2, where all registers but $(\overline{0})$ and $(\overline{1})$ are easily seen to have a constant known value in each state. Eliminating the redundant registers means that we only have to maintain two registers at run-time.

We achieve this by constant propagation: computing reaching definitions by solving a set of data-flow constraints (see e.g. [**?**]).

5.4 Symbolic representation

Text transformation programs often contain idioms which have a rather redundant representation as pure transducers. A program might for example match against a whole range of characters and proceed in the same way regardless of which one was matched. This will, however, lead to a transition for each concrete character in the generated FST, even though all transitions have the same source and destination states.

A more succinct representation can be obtained by using a symbolic representation of the transition relation by introducing transitions whose input labels are *predicates*, and whose output labels are *terms* indexed by input symbols. Replacing input labels with predicates has been described first described by Watson [**?**]. Such symbolic transducers have been developed

further and have recently received quite a bit of attention, with applications in verification and verifiable string transformations [**?**, **?**, **?**, **?**].

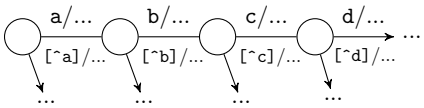
Our implementation of Kleenex uses a symbolic representation for basic ranges of symbols in order to get rid of most redundancies. The simulation algorithm and the SST construction can be generalized to the symbolic case without altering the fundamental structure, so we have elided the details of this optimization. We refer the reader to the cited literature for the technical details of symbolic transducers.

5.5 Finite lookahead

A common pattern in Kleenex programs are definitions of the form

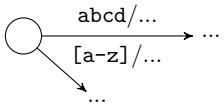
```
token := ~/abcd/ commonCase | ~/[a-z]+/ fallback
```

that is, a specific pattern appearing with higher priority than a more general fallback pattern. Patterns of this form will result in (symbolic) SSTs containing the following kind of structure:



The primary case and the fallback pattern are simulated in lockstep, and in each state there is a transition for when the common case fails after reading 0, 1, 2, etc. symbols.

If the SST was able to look more than one symbol ahead before determining the next state, we would be able to tabulate a much coarser set of simulation states and do away with the fine-grained interleaving. For the above example, we would like a transition structure like the following:



If the first four symbols of the input are `abcd`, the upper transition is taken. If this is not the case, but the first symbol is `a`, then the lower transition is taken. The idea is that any string successfully matched by the primary case will satisfy the test `abcd`, so if the transition with `[a-z]` is taken, then the FST states corresponding to the primary case can be removed from the generalized state set and tabulation can continue with a simpler simulation state.

The semantics of SSTs with lookahead are still deterministic despite the seeming overlap of patterns, as the model requires that any pair of tests are either disjoint (no string will satisfy both at the same time), or one test is completely contained in another (if a string satisfies the first test, it also satisfies the second). This restriction gives a total order between tests, specifying their priority—the most specific test must be tried first.

6 Benchmarks

We have run comparisons with different combinations of the following tools:

RE2, Google's automata-based regular expression C++ library [**?**].
RE2J, a recent re-implementation of RE2 in Java [**?**].
GNU `AWK`, GNU `grep`, and GNU `sed`, programming languages and tools for text processing and extraction [**?**].
Oniglib, a regular expression library written in C++ with support for different character encodings [**?**].
Ragel, a finite state machine compiler with multiple language backends [**?**].

In addition, we implemented test programs using the standard regular expression libraries in the scripting languages Perl [**?**], Python [**?**], and Tcl [**?**].

Meaning of plot labels Kleenex plot labels indicate the compilation path, and follow the format `[<0|3>[-la] | woACT] [clang|gcc]`. `0/3` indicates whether constant propagation was disabled/enabled. `la` indicates whether lookahead was enabled. `clang`/`gcc` indicates which C compiler was used. The last part indicates that custom register updates are disabled, in which case we generate a single fused SST as described in 6.3. These are only run with constant propagation and lookahead enabled.

Experimental setup The benchmark machine runs Linux, has 32 GB RAM and an eight-core Intel Xeon E3-1276 3.6 GHz CPU with 256 KB L2 cache and 8 MB L3 cache. Each benchmark program was run 15 times, after first doing two warm-up rounds. Version numbers of libraries, etc. are included in the appendix. All C and C++ files have been compiled with `-O3`.

Difference between Kleenex and the other implementations Unless otherwise stated, the structure of all the non-Kleenex implementations is a loop that reads input line by line and applies an action to the line. Hence, in these implementations there is an interplay between the regular expression library used and the external language, e.g., RE2 and C++. In Kleenex, line breaks do not carry any special significance, so the multi-line programs can be formulated entirely within Kleenex.

Ragel optimization levels Ragel is compiled with three different optimization levels: T1, F1, and G2. "T1" and "F1" means that the generated C code should be based on a lookup-table, and "G2" means that it should be based on C `goto` statements.

Kleenex compilation timeout On some plots, some versions of the Kleenex programs are not included. This is because the C compiler has timed out (after 30 seconds). As we fully determinize the transducers, the resulting C code can explode in some cases. This is a an area for future research.

6.1 Baseline

The following three programs are intended to give a baseline impression of the performance of Kleenex programs.

`flip_ab` The program `flip_ab` swaps "a"s and "b"s on all its input lines. In Kleenex it looks like this:

```
main := ("b" ~/a/ | "a" ~/b/ | /\n/)*
```

We made a corresponding implementation with Ragel, using a `while`-loop in C to get each new input line and feed it to the automaton code generated by Ragel.

Implementing this functionality with regular expression libraries in the other tools would be an unnatural use of them, so we have not measured those.

The performance of the two implementations run on input with an average line length of 1000 characters is shown in Figs. 4.

`patho2` The program `patho2` forces Kleenex to wait until the very last character of each line has been read before it can produce any output:

```
main := ((~/[a-z]*a/ | /[a-z]*b/)? /\n/)+
```

In this benchmark, the constant propagation makes a big difference, as Fig. 5 shows. Due to the high degree of interleaving and the lack of keywords, in this program the look-ahead optimization reduces overall performance.

This benchmark was not run with Ragel because Ragel requires the programmer to do all disambiguation manually when writing the program; the C code that Ragel generates does not handle ambiguity in a predictable way.

6.2 Rewriting

Thousand separators The following Kleenex program inserts thousand separators in a sequence of digits:

```
main  := (num /\n/)*
num   := digit{1,3} ("," digit{3})*
digit := /[0-9]/
```

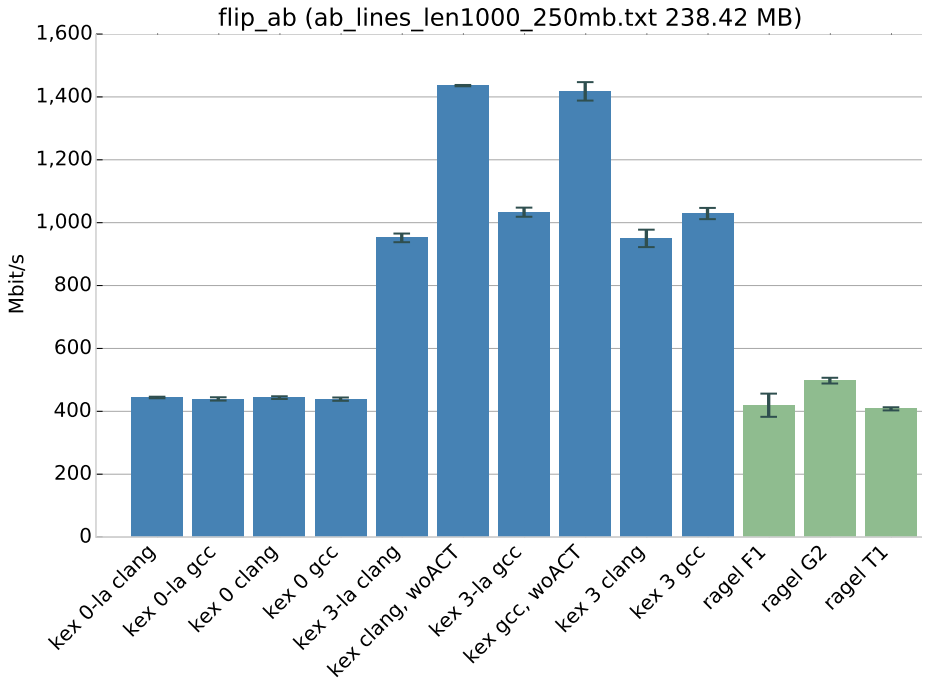


Figure 4: `flip_ab` run on lines with average length 1000.

We evaluated the Kleenex implementation along with three other implementations using Perl, and Python. The performance can be seen in Fig. 6. Both Perl and Python are significantly slower than all of the Kleenex implementations; this is a problem that is tricky to formulate with normal regular expressions (unless one reads the input right-to-left).

CSV rewriting The program `csv_project3` deletes columns two and five from a CSV file:

```
main := (row /\n/)*
col  := /[^,\n]*/
row  := ~(col /,/) col "\t" ~/,/ ~(col /,/)
        ~(col /,/) col ~/,/      ~col
```

Various specialized tools exist that can handle this transformation are included in Fig. 7; GNU `cut` is a command that splits its input on certain characters, and GNU `AWK` has built-in support for this type of transformation.

Apart from `cut`, which is really fast for its own use-case, the Kleenex implementation is the fastest. The performance of Ragel is slightly lower, but this is likely due to the way the implementation produces output: In a Kleenex program, output strings are automatically put in an output buffer

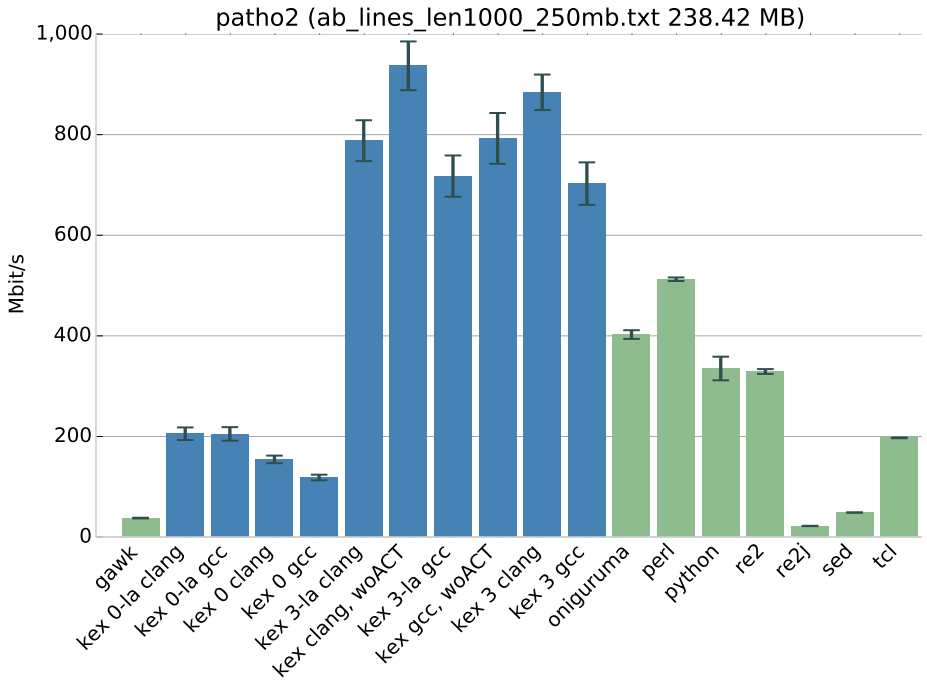


Figure 5: `patho2` run on lines with average length 1000.

which is flushed routinely, whereas a programmer has to manually handle buffering when writing a Ragel program.

IRC protocol handling The following Kleenex program parses the IRC protocol as specified in RFC 2812.[4] It follows roughly the output style described in part 2.3.1. Note that the Kleenex source code and the BNF grammar in the RFC are almost identical. Fig. 8 shows the throughput on 250 MiB data.

```
main := (message | "Malformed line: " /[^\r\n]*\r?\n/)*
message := (~/:/ "Prefix: " prefix "\n"  ~/ /)?
           "Command: " command "\n"
           "Parameters: " params? "\n"
           ~crlf
command := letter+ | digit{3}
prefix := servername
        | nickname ((/!/ user)? /@/ host )?
user := /[^\n\r @]/+ // Missing \x00
middle := nospcrlfcl ( /:/ | nospcrlfcl )*
params := (~/ / middle ", "){,14} ( ~/ :/ trailing )?
        | ( ~/ / middle ){14} ( / / /:/?  trailing )?
```

[4] `https://tools.ietf.org/html/rfc2812`

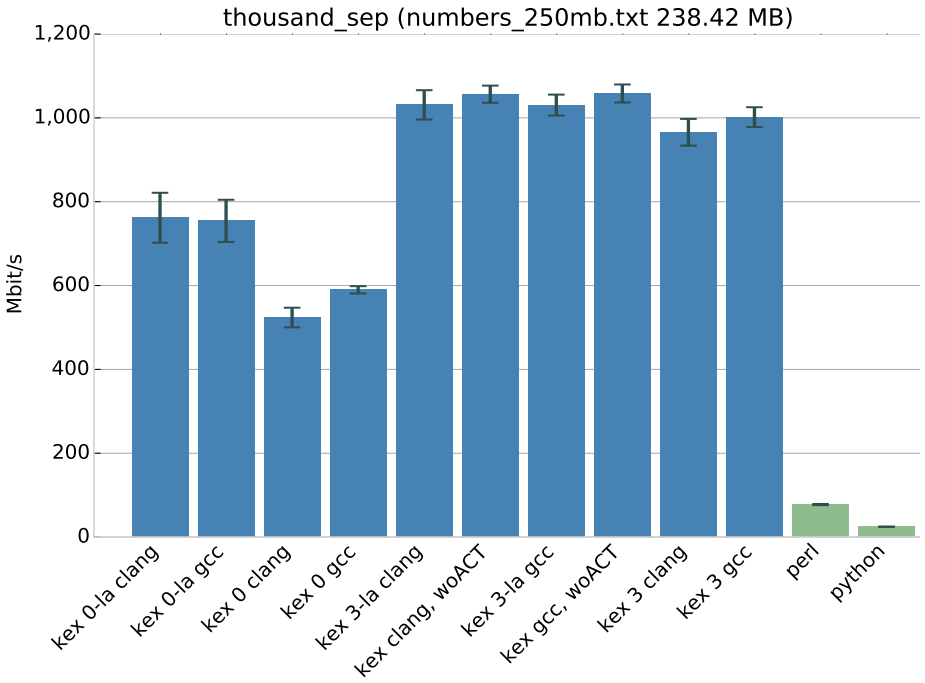


Figure 6: Inserting thousand separators on random numbers with average length 1000.

```
trailing := (/:/ | / / | nospcrlfcl)*
nickname := (letter | special)
            (letter | special | digit){,10}
host := hostname | hostaddr
servername := hostname
hostname := shortname ( /\./ shortname)*
hostaddr := ip4addr
shortname := (letter | digit) (letter | digit | /-/)*
             (letter | digit)*
ip4addr := (digit{1,3} /\./ ){3} digit{1,3}
```

6.3 With or without action-separation

One can choose to use the machine resulting in combining the oracle and the action machine when compiling Kleenex. Doing so results in only one process doing both the disambiguation and outputting, which in some cases is faster and in other slower. Figs. 7, 9, and 11 illustrate both situations. It depends on the structure of the problem whether it pays off to split up the work in two processes; if all the work happens in the oracle and the action machine nearly does not do anything, then the added overhead incurred by the process context switches becomes noticeable. On the other hand, in

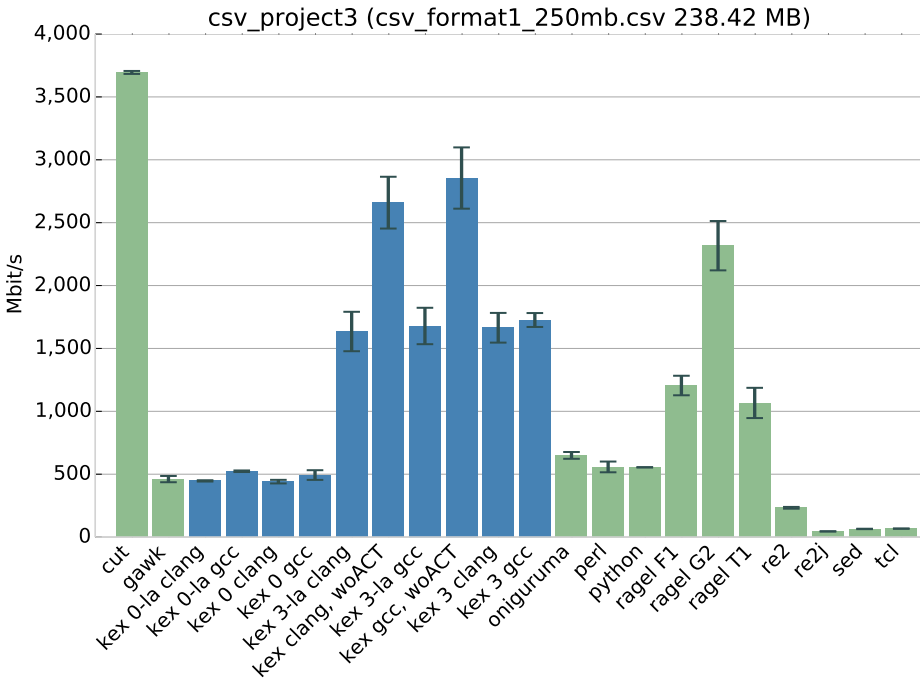


Figure 7: `csv_project3` reads in a CSV file with six columns and outputs columns two and five. "gawk" is GNU `AWK` that uses the native `AWK` way of splitting up lines. "cut" is a command from GNU coreutils that splits up lines.

cases where both machines do much work, the fact that two CPU cores can be utilized speeds up the program. This would be more likely if Kleenex had support for actions which could perform arbitrary computation, e.g. in the form of embedded C code.

7 Use cases

In this section we will briefly touch upon various interesting use cases for Kleenex.

JSON logs to SQL We have implemented a Kleenex program (code in Appendix) that transforms a JSON log file into an SQL insert statement. The program works on the logs provided by Issuu.[5]

The Ragel version we implemented outperforms Kleenex by about 50% (Fig. 9), indicating that further optimizations of our SST construction should be possible.

[5]The line-based data set consists of 30 compressed parts and part one is available from `http://labs.issuu.com/anodataset/2014-03-1.json.xz`

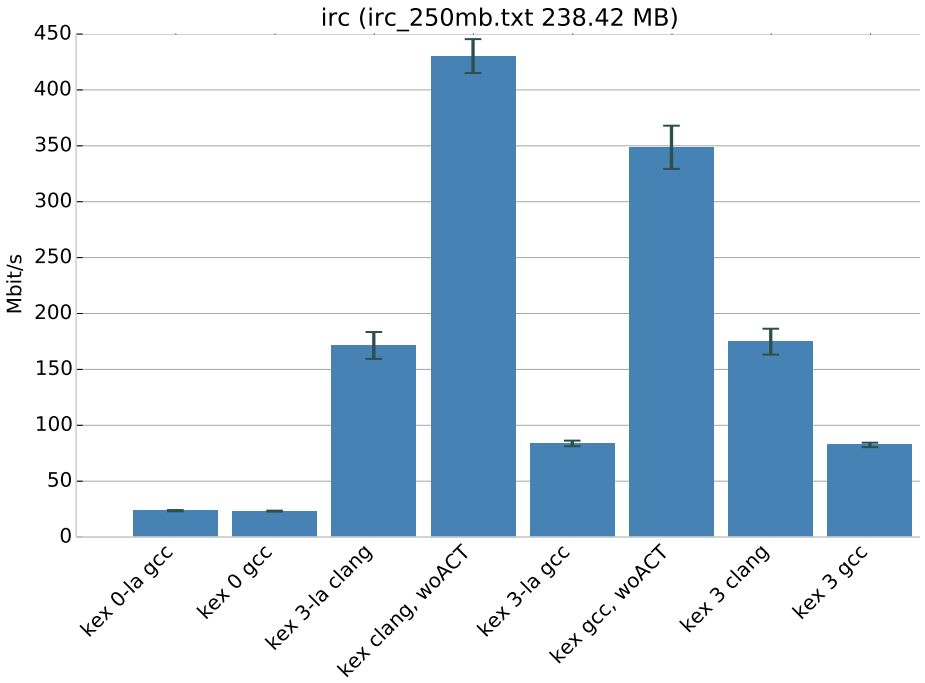


Figure 8: Throughput when parsing 250 MiB random IRC data.

Apache CLF to JSON The Kleenex program below rewrites Apache CLF[6] log files into a list of JSON records:

```
main := "[" loglines? "]\n"
loglines := (logline "," /\n/)* logline /\n/
logline := "{" host ~sep ~userid ~sep ~authuser sep
               timestamp sep request sep code sep
               bytes sep referer sep useragent "}"
host := "\"host\":\"" ip "\""
userid := "\"user\":\"" rfc1413 "\""
authuser := "\"authuser\":\"" /[^ \n]+/ "\""
timestamp := "\"date\":\"" ~/\[/ /[^\n\]]+/ ~/]/ "\""
request := "\"request\":" quotedString
code := "\"status\":\"" integer "\""
bytes := "\"size\":\"" (integer | /-/) "\""
referer := "\"url\":" quotedString
useragent := "\"agent\":" quotedString
ws := /[\t ]+/
sep := "," ~ws
quotedString := /"([^"\n]|\\")*"/
integer := /[0-9]+/
ip := integer (/\./ integer){3}
```

[6] https://httpd.apache.org/docs/trunk/logs.html#common

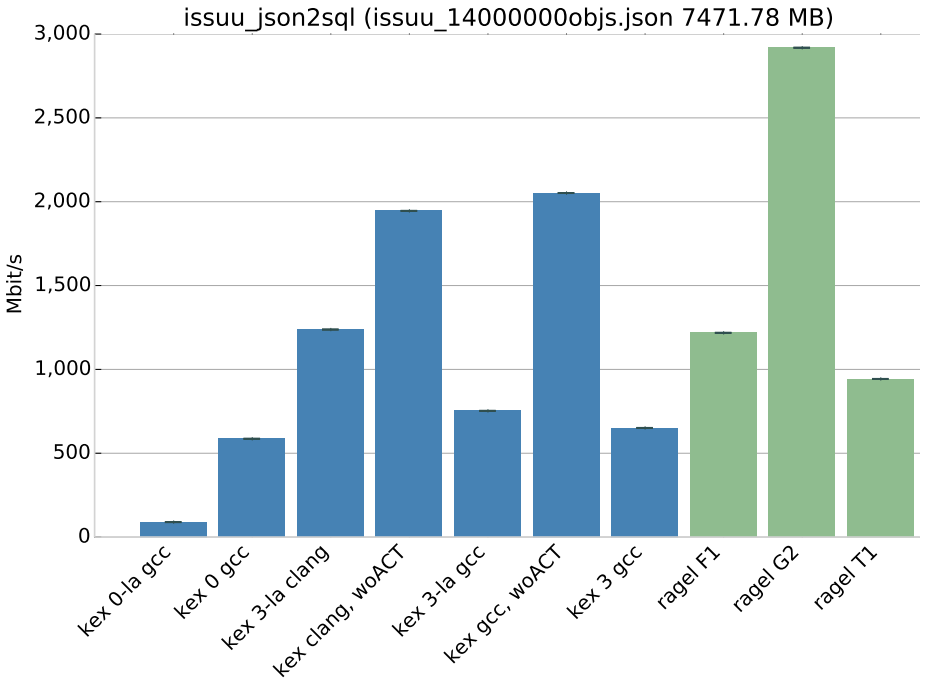


Figure 9: The speeds of transforming JSON objects to SQL INSERT statements using Ragel and Kleenex.

```
rfc1413 := /-/
```

This is a re-implementation of a Ragel program.[7] Fig. 10 shows the benchmark results. The versions compiled with clang are not included, as the compilation timed out after 30 seconds. Curiously, the non-optimized Kleenex program is the fastest in this case.

ISO date/time objects to JSON Inspired by an example in [**?**], the program `iso_datetime_to_json` (code in Appendix) converts date and time stamps in an ISO standard format to a JSON object. Fig. 11 shows the performance.

URL parsing Kleenex allows one to naturally follow the URL specification given in RFC1738.[8] We implemented a URL parser by directly following the BNF-grammar in the RFC; its code can be found in the Appendix.

Syntax highlighting Kleenex can used to write syntax highlighters; in fact, the Kleenex syntax in this paper was highlighted with a Kleenex pro-

[7] `https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel`
[8] `http://www.ietf.org/rfc/rfc1738.txt`

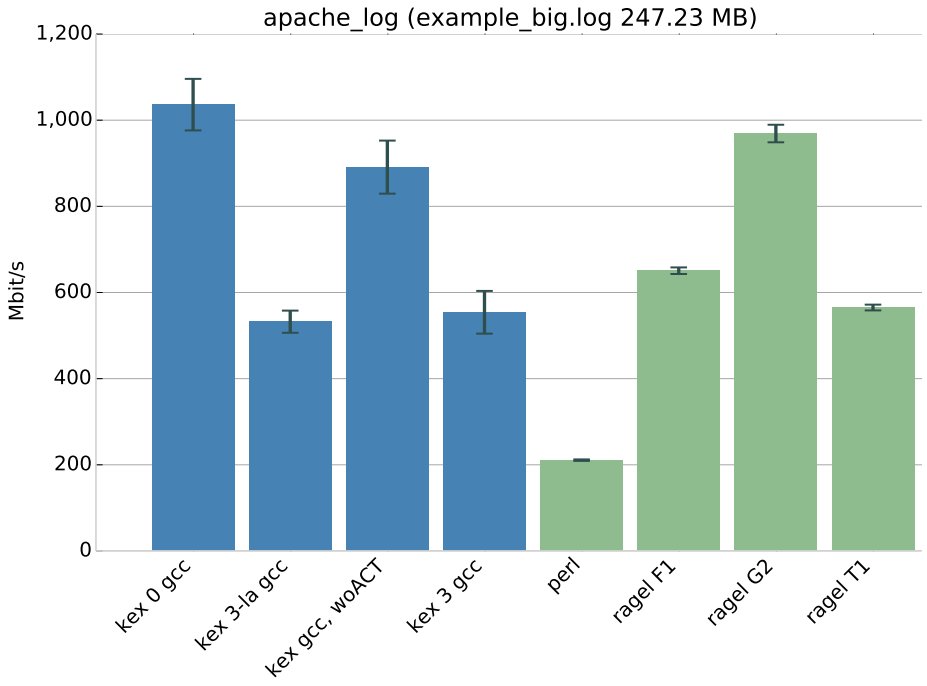


Figure 10: Speed of the conversion from the Apache Common Log Format to JSON.

gram. The code for a version that emits ANSI color codes is included in the Appendix.

HTML comments The following Kleenex program finds HTML comments with basic formatting commands and renders them in HTML after the comment. For example, `<!-- doc:  *Hello* world -->` becomes `<!-- doc:  *Hello* world --><div> <b>Hello</b> world </div>`.

```
main := (comment | /./)*
comment := /<!-- doc:/ clear doc* !orig /-->/
           "<div>" !render "</div>"
doc := ~/\*/ t@/[^*]*/ ~/\*/
       [ orig += "*" t "*" ] [ render += "<b>" t "</b>" ]
     | t@/./ [ orig += t ] [ render += t ]
clear := [ orig  <- "" ] [ render <- "" ]
```

8 Related Work

We discuss related work in the context of current and future work.

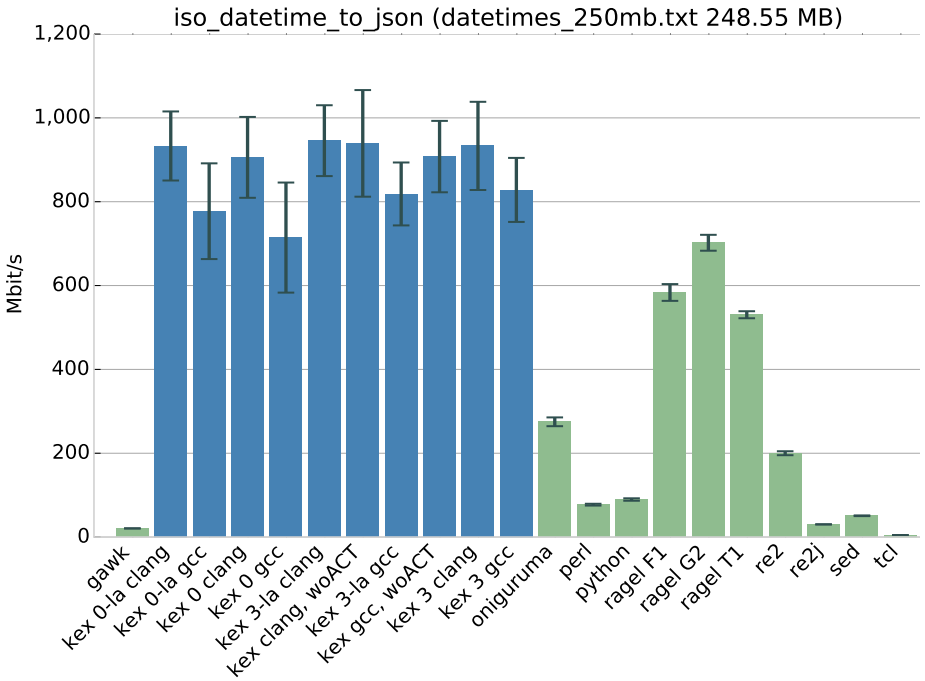


Figure 11: The performance of the conversion of ISO time stamps into JSON format.

8.1 Regular expression matching

Regular expression *matching* has different meanings in the literature.

For *acceptance testing*, which corresponds to classical automata theory, Bille and Thorup [**?**] improve on Myers' [**?**] log-factor improved RE-membership testing of classical NFA-simulation, based on tabling. They design an $O(kn)$ algorithm [**?**] with word-level parallelism, where $k \leq m$ is number of strings occurring in an RE. The tabling technique may be promising in practice; the algorithms have not been implemented and evaluated empirically, though.

In *subgroup matching* as in PCRE [**?**], an input is not only classified as accepting or not, but a substring is returned for each sub-RE in an RE designated to be of interest. Subgroup matching is often implemented by backtracking over alternatives, which yields the greedy match.[9] It may result in exponential-time behavior, however. Consequently, considerable human effort is expended to engineer REs to perform well. REs resulting in exponential run-time behavior are used in algorithmic attacks, leading to proposals for countermeasures to such attacks by classifying REs with

[9]Committing to the left alternative before checking that the remainder of the input is accepted is the essence of parsing expression grammars [**?**].

slow backtracking performance [?, ?], where the countermeasures in turn appear to be attackable. Even in the absence of inherently hard matching with backreferences [?], backtracking implementations with avoidable performance blow-ups are amazingly wide-spread. This may be due to a combination of their good best-case performance and PCRE-embellishments driven by use cases. Some submatch libraries with guaranteed worst-case linear-time performance, notably RE2 [?], are making inroads, however. Myers, Oliva and Guimaraes [?] and Okui, Suzuki [?] describe a $O(mn)$, respectively $O(m^2n)$ POSIX-disambiguated matching algorithms. Sulzmann and Lu [?] use Brzozowski [?] and Antimirov derivatives [?] for Perl-style subgroup matching for greedy and POSIX disambiguation.

Full RE *parsing* generalizes submatching: it returns a list of matches for each Kleene-star, also for nested ones. Kearns [?], Frisch and Cardelli [?] devise 3-pass linear-time *greedy* RE parsing; they require 2 passes over the input, the first consisting of reversing the entire input, before generating output in the third pass. Grathwohl, Henglein, Nielsen, Rasmussen devise a two-pass [?] and an optimally streaming [?] greedy regular expression parsing algorithm. Streaming guarantees that line-by-line RE matching can be coded as a single RE matching problem. Sulzman and Lu [?] remark that POSIX is notoriously difficult to implement correctly and show how to use Brzozowski derivatives [?] for POSIX RE parsing;

There are specialized RE matching tools and techniques too numerous to review comprehensively. We mention a few employing automaton optimization techniques applicable to Kleenex, but presently unexplored. Yang, Manadhata, Horne, Rao, Ganapathy [?] propose an OBDD representation for subgroup matching and apply it to intrusion detection REs; the cycle counts per byte appear a bit high, but are reported to be competitive with RE2. Sidhu and Prasanna [?] implement NFAs directly on an FPGA, essentially performing NFA-simulation in parallel; it outperforms GNU `grep`. Brodie, Taylor, Cytron [?] construct a multistride DFA, which processes multiple input symbols in parallel, and devise a compressed implementation on stock FPGA, also achieving very high throughput rates. Likewise, Ziria employs tabled multistriding to achieve high throughput [?]. Navarro and Raffinot [?], show how to code DFAs compactly for efficient simulation.

8.2 Ambiguity

REs may be ambiguous, which is irrelevant for acceptance testing, but problematic for submatching and parsing since the output depends on which amongst possibly multiple matches is to be returned. Brüggemann-Klein [?] provides an efficient $O(m^2)$ RE ambiguity testing algorithm. Vansummeren [?] illustrates differences between POSIX, first/longest and greedy matches. Colcombet [?] analyzes notions of (non)determinism of automata.

8.3 Transducers

From RE parsing it is a surprisingly short distance to the implementation of arbitrary nondeterministic finite state transducers (NFSTs) [**?**, **?**]. In contrast to the situation for *automata*, nondeterministic transducers are strictly more powerful than deterministic transducers; this, together with observable ambiguity, highlights why RE parsing is more challenging than RE acceptance testing.

As we have seen, efficient RE parsing algorithms operate on arbitrary NFAs, not only those corresponding to REs. Indeed, REs are not a particularly convenient or compact way of specifying regular languages: they can be represented by *certain* small NFAs with low tree-width, but may be inherently quadratically bigger even for DFAs [**?**, Theorem 23]. This is why Kleenex employs context-free grammars restricted to denote regular languages, with embedded output actions, to denote NFSTs.

We have shown that NFSTs, in particular unambiguous NFSTs, can be implemented by a *subclass* of streaming string transducers (SSTs). SSTs extensionally correspond to regular transductions, functions implementable by 2-way deterministic finite-state transducers [**?**], MSO-definable string transductions [**?**] and a combinator language analogous to regular expressions [**?**]. The implementation techniques used in Kleenex appear to be directly applicable to all SSTs, not just the ones corresponding to NFSTs.

Allender and Mertz [**?**] show that the functions computable by register automata, which generalize output strings to arbitrary monoids, are in NC and thus inherently parallelizable. This is achievable by performing relational NFST-composition by matrix multiplication on the matrix representation of NFSTs [**?**], which can be performed by parallel reduction. This is tantamount to running an NFST from all states, not just the input state, on input string fragments. Mytkowicz, Musuvathi, Schulte [**?**] observe that there is often a small set of cut states sufficient to run each NFST. This promises to be an interesting parallel harness for a suitably adapted Kleenex implementation running on fragments of very large inputs.

Veanes, Molnar, Mytkowics [**?**] employ symbolic transducers [**?**, **?**, **?**] and a data-parallel intermediate language in the implementation of BEK for multicore execution.

9 Conclusions

We have presented Kleenex, a convenient language for specifying nondeterministic finite state transducers; and its compilation to machine code representations of streaming state transducers, which emit the output

Kleenex is comparatively expressive and performs consistently well—for complex regular expressions with nontrivial amounts of output almost

always better in the evaluated use cases—vis-à-vis text processing tools such as RE2, Ragel, `grep`, `AWK`, `sed` and RE-libraries of Perl, Python and Tcl.

We believe Kleenex's clean semantics, streaming optimality, algorithmic generality, worst-case guarantees and absence of tricky code and special casing provide a useful basis for

- extensions to deterministic visible push-down automata, restricted versions of backreferences and approximate/probabilistic matching;
- known, but so far unexplored optimizations, such as multicharacter input processing, automata minimization and symbolic representation, hybrid NFST-simulation/SST-construction (analogous to NFA-simulation with NFA-state set memoization to implement on-demand DFA-construction);
- massively parallel (log-depth, linear work) input processing.

Acknowledgements

We thank Issuu for releasing their data set to the research community. This work has been partially supported by The Danish Council for Independent Research under Project 11-106278, "Kleene Meets Church: Regular Expressions and Types". The order of authors is insignificant; please list all authors—or none—when citing this paper.