Kleenex: Compiling Nondeterministic Transducers to Deterministic Streaming Transducers

Bjørn Bugge Grathwohl Fritz Henglein
Ulrik Terp Rasmussen
DIKU, University of Copenhagen, Denmark
{bugge,henglein,dolle}@diku.dk

Kristoffer Aalund Søholm
Sebastian Paaske Tørholm
Jobindex, Denmark
{soeholm,sebbe}@diku.dk

Abstract

We present and illustrate Kleenex, a language for expressing general nondeterministic finite transducers, and its novel compilation to streaming string transducers with worst-case linear-time performance and sustained high throughput. Its underlying theory is based on transducer decomposition into oracle and action machines: the oracle machine performs streaming greedy disambiguation of the input; the action machine performs the output actions. In use cases Kleenex achieves consistently high throughput rates around the 1 Gbps range on stock hardware. It performs well, especially in complex use cases, in comparison to both specialized and related tools such as AWK, sed, RE2, Ragel and regular-expression libraries.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.2 [*Language Classifications*]: Specialized application languages; F.1.1 [*Models of Computation*]: Automata

Keywords regular, automaton, nondeterministic, transducer, determinization, streaming

1. Introduction

A Kleenex program consists of a context-free grammar, restricted to guarantee regularity, with embedded side-effecting semantic actions.

We illustrate Kleenex by an example. Consider a large text file containing unbounded numerals, which we want to make more readable by inserting separators; e.g. “12742” is to be replaced by “12,742”). In Kleenex, this transformation can be specified as follows:

```
main  := (num /[^0-9]/ | other)*
num   := digit{1,3} ("," digit{3})*
digit := /[0-9]/
other := /./
```

This is the complete program. The program defines a set of nonterminals, with `main` being the start symbol. The constructs `/[0-9]/`, `/[^0-9]/` and `/./` specify matching a single digit, any non-digit and any symbol, respectively, and echoing the matched symbol to the output. The construct `","` reads nothing and outputs a single comma. The star `*` performs the inner transformation zero or more times; the repetition `{1,3}` performs it between 1 and 3 times. Finally, the `|` operator denotes prioritized choice, with priority given to the left alternative. An example of its execution is as follows:

Input read so far	. . . and output produced so far
Surf	Surf
Surface:␣	Surface:␣
Surface:␣14479	Surface:␣
Surface:␣1447985	Surface:␣
Surface:␣144798500␣	Surface:␣144,798,500␣
Surface:␣144798500␣km^2	Surface:␣144,798,500␣km^2

The example highlights the following:

Ambiguity by design. Any string is *accepted* by this program, since any string matching `num /[^0-9]/` also matches `(other)*`. Greedy disambiguation forces the `num /[^0-9]/` transformation to be tried first, however, and only if that fails do we fall back to echoing the input verbatim to the output using `other`.

Streaming output. The program almost always detects the earliest possible time an output action can be performed. Any non-digit symbol is written to the output immediately, and as soon as the first non-digit symbol after a sequence of digits is read, the resulting numeral with separators is written to the output stream. The first of a sequence of digits is not output right away, however. Employing a strategy that *always* outputs as early as possible would require solving a PSPACE-hard problem.

A Kleenex program is first compiled to a possibly ambiguous *(finite-state) transducer*. Any transducer can be decomposed into two transducers: an *oracle machine*, which maps an input string to a bit-coded representation of the transducer paths accepting the input, and a deterministic *action machine*, which translates such a bit-code to the corresponding sequence of output actions in the original transducer. The greedy leftmost path in the oracle machine corresponds to the lexicographically least bit-code of paths accepting a given input; consequently, disambiguation reduces to computing this bit-code for a given input. To compute it, the oracle machine is simulated in a streaming fashion. This generalizes NFA simulation to not just yield a single-bit output—accept or reject—but also the lexicographically least path witnessing acceptance. The simulation algorithm maintains a *path tree* from the initial state to all the oracle machine states reachable by the input prefix read so far. A branching node represents both sides of an alternative where both are still viable. The output actions on the (possibly empty) path segment from the initial state to the first branching node can be performed based on the input prefix processed so far without knowing which of the presently reached states will eventually accept

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837647

the rest of the input. This algorithm generalizes *greedy regular expression parsing* [31, 32] to *arbitrary* right-regular grammars. Regular expressions correspond to certain well-structured oracle machines via their McNaughton-Yamada-Thompson construction. The simulation algorithm *automatically* results in constant memory space consumption for grammars that are deterministic modulo finite lookahead, e.g. one-unambiguous regular expressions [19]. For arbitrary transducers the simulation requires linear space in the size of the input in the worst case. No algorithm can guarantee constant space consumption: the number of unique path trees computed by the streaming algorithm is potentially unbounded due to the possibility of arbitrarily much lookahead required to determine which of two possible alternatives will eventually succeed. Unbounded lookahead is the reason that not all unambiguous transducers can be determinized to a finite state machine [13, 53].

By identifying path trees with the same ordered leaves and underlying branching structure, we obtain an equivalence relation with finite index. That is, a path tree can be seen as a rooted full binary tree together with an association of output strings with tree edges, and the set of reachable rooted full binary trees of an oracle machine can can be precomputed analogous to the NFA state sets reachable in an NFA. We can thus compile an oracle machine to a *streaming string transducer* [4, 5, 9], a deterministic machine model with (unbounded sized) string registers and affine (copy-free) updates associated with each transition: a path tree is represented as an abstract state and the contents of a finite set of registers, each containing a bit sequence coding a path segment of the represented path tree. Upon reading an input, the state is changed and the registers are updated in-place to represent the subsequent path tree. This yields a both asymptotically and practically very efficient implementation: the example shown earlier compiles to an efficient C program that operates with sustained high throughput in the 1 Gbps range on stock desktop hardware.

The semantic model of context-free grammars with unbridled "regular" ambiguity and embedded semantic actions is flexible and the above implementation technology is quite general. For example, the action transducer is not constrained to producing output in the string monoid, but can be extended to any monoid. By considering the monoid of affine register updates, Kleenex can code all nondeterministic streaming string transducers [6].

1.1 Contributions

This paper makes the following novel contributions:

- A *streaming* algorithm for *nondeterministic finite state transducers (FST)*, which emits the lexicographically least output sequence generated by all accepting paths of an input string based on decomposition into an input-processing *oracle machine* and an output-effecting *action machine*. It runs in $O(mn)$ time for transducers of size m and inputs of size n.
- An effective determinization of FSTs into a subclass of *streaming string transducers (SST)* [4], finite state machines with copy-free updating of string registers when entering a new state upon reading an input symbol.
- An expressive declarative language, *Kleenex*, for specifying FSTs 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.[1] Basic Kleenex programs can be functionally composed into pipelines. The central technical aspect of Kleenex is its semantic support for unbridled nondeterminism and its effective determinization and compilation to SSTs, which both highlights and complements the significance of SSTs as a deterministic machine model.
- An implementation, including empirically evaluated optimizations, of Kleenex that generates SSTs and deterministic finite-state machines, each rendered as standard single-threaded C-code that is eventually compiled to x86 machine code. The optimizations illustrate the design and implementation flexibility obtained by the underlying theories of FSTs and SSTs.
- Use cases that illustrate the expressive power of Kleenex, and a performance comparison with related tools, including Ragel [65], RE2 [62] 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 less expressive tools with special-cased algorithms and to tools with no or limited support for nondeterminism.

1.2 Overview of paper

In Section 2 we introduce normalized transducers with explicit deterministic and nondeterministic ϵ-transitions. Kleenex and its translation to such transducers is defined in Section 3. We then devise an efficient streaming transducer simulation (Section 4) and its determinization (Section 5) to streaming string transducers. In Section 6 we briefly describe the compilation to C-code and some optimizations, and we then empirically evaluate the implementation on a number of simple benchmarks and more realistic use cases (Section 7). We conclude with a discussion of related and possible future work (Section 8).

We assume basic knowledge of automata [39], compilation [2], and algorithms [21]. Basic results in these areas are not explicitly cited.

2. Transducers

An *alphabet* A is a finite set; e.g. the binary alphabet $\mathbf{2} = \{0, 1\}$ and the empty alphabet $\emptyset = \{\}$. A^* denotes the free monoid generated by A, that is the strings over A with concatenation, expressed by juxtaposition, and the empty string ε as neutral element. We write $A[x, \ldots]$ for extending A with additional elements $x, \ldots$ not in A.

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

- Σ and Γ are alphabets;
- Q is a finite set of *states*;
- $q^-, q^f \in Q$ are the *initial* and *final* states, respectively;
- $E : Q \times \Sigma[\epsilon] \times \Gamma[\epsilon] \times Q$ is the *transition relation*.

Its *size* is the cardinality of its transition relation: $|T| = |E|$.

$\mathcal{T}$ is *deterministic* if for all $q \in Q, a \in \Sigma[\epsilon]$ we have

$$(q, a, b', q') \in E \wedge (q, a, b'', q'') \in E \quad \Rightarrow \quad b' = b'' \wedge q' = q''$$
$$(q, \epsilon, b', q') \in E \wedge (q, a, b'', q'') \in E \quad \Rightarrow \quad \epsilon = a$$

The *support* of a state is the set of symbols it has transitions on:

$$\mathrm{supp}(q) = \{a \in \Sigma[\epsilon] \mid \exists q', b. (q, a, b, q') \in E\}.$$

Deterministic FSTs with no ϵ-transitions and $\mathrm{supp}(q) = \Sigma$ for all q are *Mealy machines*. Conversely, every deterministic FST is easily turned into a Mealy machine by adding a failure state and transitions to it.

We write $q \xrightarrow{a/b} q'$ whenever $(q, a, b, q') \in E$, and E is understood from the context. A *path* in $\mathcal{T}$ is a possibly empty sequence of transitions

$$q_0 \xrightarrow{a_1/b_1} q_1 \xrightarrow{a_2/b_2} \ldots \xrightarrow{a_n/b_n} q_n$$

It has *input* $u = a_1 a_2 \ldots a_n$ and *output* $v = b_1 b_2 \ldots b_n$. We write $q_0 \xrightarrow{u/v} q_n$ if there exists such a path.

[1] This avoids the $\Omega(M(n))$ lower bound for context-free grammar parsing, where $M(n)$ is the complexity of multiplying $n \times n$ matrices [40].

Definition 2 (Relational semantics, input language)**.** FST $\mathcal{T}$ denotes the binary relation

$$\mathcal{R}[\![\mathcal{T}]\!] = \{(\overline{u}, \overline{v}) \mid q^- \xrightarrow{u/v} q^f\}$$

where the *ϵ-erasure* $\bar{\cdot} : \Sigma[\epsilon]^* \to \Sigma^*$ is $\overline{\epsilon} = \varepsilon$ and $\overline{a} = a$ for all $a \in \Sigma$, extended homomorphically to strings. Its *input language* is

$$\mathcal{L}[\![\mathcal{T}]\!] = \{s \mid \exists t \,.\, (s, t) \in \mathcal{R}[\![\mathcal{T}]\!]\}.$$

Two FSTs are *equivalent* if they have the same relational semantics.

The class of relations denotable by FSTs are the *rational relations*; their input languages are the *regular languages* [13].

Definition 3 (Normalized FST)**.** A *normalized finite state transducer* over Σ and Γ is a deterministic FST over $\Sigma[\epsilon_0, \epsilon_1]$ and Γ such that for all $q \in Q$, q is:

- a *choice state*: $\mathrm{supp}(q) = \{\epsilon_0, \epsilon_1\}$ and $q \neq q^f$, or
- a *skip state*: $\mathrm{supp}(q) = \{\epsilon\}$ and $q \neq q^f$, or
- a *symbol state*: $\mathrm{supp}(q) = \{a\}$ for some $a \in \Sigma$ and $q \neq q^f$, or
- the *final state*: $\mathrm{supp}(q) = \{\}$ and $q = q^f$

We say that q is a *resting state* if q is either a symbol state or the final state.

The relational semantics $\mathcal{R}[\![\mathcal{T}]\!]$ of a normalized FST is the same as in Definition 2, where ϵ-erasure is extended by $\overline{\epsilon}_0 = \overline{\epsilon}_1 = \varepsilon$.

Proposition 1. *For every FST of size m there exists an equivalent normalized FST of size at most $3m$. Conversely, for every normalized FST of size m there exists an equivalent FST of the same size.*

Proof. (Sketch) For each state q with $k > 1$ outgoing transitions, add k new states $q^{(1)}, \ldots, q^{(k)}$, replace the i-th outgoing transition (q, a, b, q') by $(q^{(i)}, a, b, q')$ and add a full binary tree of ϵ_0- and ϵ_1-transitions for reaching each $q^{(i)}$ from q. In the converse direction, replace ϵ_0 and ϵ_1 by ϵ. □

Normalized FSTs are useful by limiting transition outdegree to 2, having explicit ϵ-transitions and classifying them into deterministic (ϵ) and ordered nondeterministic ones (ϵ_0, ϵ_1).

Proviso. Henceforth we will call normalized FSTs simply *transducers*.

Let $|\cdot| : \Sigma[\epsilon_0, \epsilon_1, \epsilon] \to \mathbf{2}[\epsilon]$ be defined by $|\epsilon_0| = 0, |\epsilon_1| = 1$ and $|a| = \epsilon$ for all $a \in \Sigma[\epsilon]$.

Definition 4 (Oracle and action machines)**.** Let $\mathcal{T}$ be a transducer. The *oracle machine* $\mathcal{T}^{\mathsf{C}}$ is defined as $\mathcal{T}$, but with each transition (q, a, b, q') replaced by $(q, a, |a|, q')$. Its *action machine* $\mathcal{T}^{\mathsf{A}}$ is $\mathcal{T}$, but with each transition (q, a, b, q') replaced by $(q, |a|, b, q')$.

The oracle machine is a transducer over Σ and $\mathbf{2}$; the action machine a deterministic FST over $\mathbf{2}$ and Γ. Each transducer can be canonically decomposed into its oracle and action machines:

Proposition 2. $\mathcal{R}[\![\mathcal{T}]\!] = \mathcal{R}[\![\mathcal{T}^{\mathsf{A}}]\!] \circ \mathcal{R}[\![\mathcal{T}^{\mathsf{C}}]\!]$

where $\circ$ denotes relational composition. Note that the oracle machine is independent of the outputs in the original transducer; in particular, a transducer where only the outputs are changed has the same oracle machine. Intuitively, the action machine starts at the initial state the original transducer, automatically follows transitions from resting and skip states, and uses the bit string from the oracle machine as an oracle—hence the name—to choose which transition to take from a choice state; in this process it emits the outputs it traverses.

Example 1. Figure 1 shows a Kleenex program (see Section 3), the associated transducer and its decomposition into oracle and action machines.

Observe that if there is a path $q \xrightarrow{u/v} q'$ then u uniquely identifies the path from q to q' in a transducer and, furthermore, in an oracle machine so does v.

We write $q \xrightarrow{u/v}_{\mathsf{np}} q''$ if the path $q \xrightarrow{u/v} q''$ does not contain an *ϵ-loop*, that is a subpath $q' \xrightarrow{u'/v'} q'$ where $\overline{u'} = \varepsilon$. Paths without ϵ-loops are called *nonproblematic* paths [29].

Definition 5 (Greedy semantics)**.** The *greedy semantics* of a transducer T is $\mathcal{G}[\![\mathcal{T}]\!] = \mathcal{R}[\![\mathcal{T}^{\mathsf{A}}]\!] \circ \mathcal{G}[\![\mathcal{T}^{\mathsf{C}}]\!]$ where

$$\begin{aligned} \mathcal{G}[\![\mathcal{T}^{\mathsf{C}}]\!] \quad = \quad & \{(\overline{u}, \overline{v}) \mid q^- \xrightarrow{u/v}_{\mathsf{np}} q^f \wedge \\ & \forall u', v' \,.\, q^- \xrightarrow{u'/v'}_{\mathsf{np}} q^f \wedge \overline{u} = \overline{u'} \Longrightarrow \overline{v} \leq \overline{v'}\} \end{aligned}$$

and $\leq$ denotes the lexicographic ordering on bit strings.

Given input string s, the greedy semantics chooses the lexicographically least path in the transducer accepting s and outputs the corresponding output symbols encountered along the path. The restriction to nonproblematic paths ensures that there are only finitely many paths accepting s and thus the lexicographically least amongst them exists, if s is accepted at all. We write $q \xrightarrow{u/v}_{\mathsf{min}} q'$ if $q \xrightarrow{u/v} q'$ is the lexicographically least nonproblematic path from q to q'.

A transducer $\mathcal{T}$ over Σ and Γ is *single-valued* if $\mathcal{R}[\![\mathcal{T}]\!]$ is a partial function from Σ^* to Γ^*.

Proposition 3. *Let $\mathcal{T}$ be a transducer over Σ and Γ.*

- *$\mathcal{G}[\![\mathcal{T}]\!]$ is a partial function from Σ^* to Γ^*.*
- *$\mathcal{G}[\![\mathcal{T}]\!] = \mathcal{R}[\![\mathcal{T}]\!]$ if $\mathcal{T}$ is single-valued.*

The greedy semantics can be thought of as a *disambiguation policy* for transducers that conservatively extends the standard semantics for single-valued transducers to a deterministic semantics for arbitrary transducers.

3. Kleenex

Kleenex[2] is a language for compactly and conveniently expressing transducers.

3.1 Core Kleenex

Core Kleenex is a grammar for directly coding transducers.

Definition 6 (Core Kleenex syntax)**.** A *Core Kleenex* program is a nonempty list $p = d_0 d_1 \ldots d_n$ of *definitions* d_i, each of the form $N := t$, where N is an identifier and t is generated by the grammar

$$t ::= \varepsilon \mid N \mid a\, N' \mid \text{"}b\text{"}\; N' \mid N_0 \,|\, N_1$$

where $a \in \Sigma$ and $b \in \Gamma$ for given alphabets Σ, Γ, e.g. some character set. N ranges over some set of identifiers. The identifiers occurring in p are called *nonterminals*. There must be at most one definition of each nonterminal, and every occurrence of a nonterminal must have a definition.

Definition 7 (Core Kleenex transducer semantics)**.** The *transducer associated with Core Kleenex program p* for nonterminal $N \in \mathcal{N}$ is

$$\mathcal{T}_p(N) = (\Sigma, \Gamma, \mathcal{N}[q^f], N, q^f, E)$$

[2] Kleenex is a contraction of *Kleene* and *expression* in recognition of the fundamental contributions by Stephen Kleene to language theory.

```
main := (num /\n/)*
num := digit{1,3}
       ("," digit{3})*
digit := /a/
```

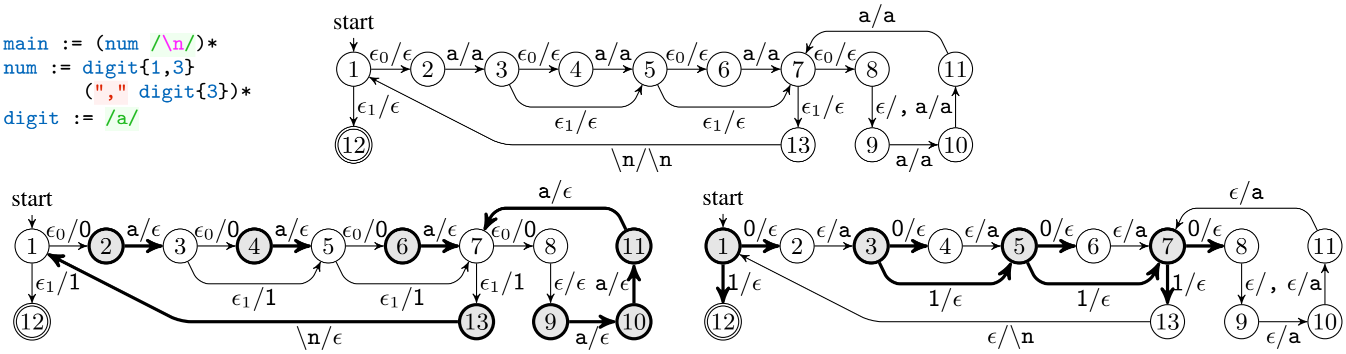

Figure 1: Top: a Kleenex program and its associated transducer. The program accepts a list of newline-separated numbers (simplified to unary numbers with digit a) and inserts thousands separators. Bottom: The corresponding oracle and action machines.

where $\mathcal{N}$ is the set of nonterminals in p, and E consists of transitions constructed from each production in p as follows:

$N{:=}\,\varepsilon$	$N \xrightarrow{\epsilon/\epsilon} q^f$
$N{:=}\,N'$	$N \xrightarrow{\epsilon/\epsilon} N'$
$N{:=}\,a\;N'$	$N \xrightarrow{a/\epsilon} N'$
$N{:=}$ "b" N'	$N \xrightarrow{\epsilon/b} N'$
$N{:=}\,N' \mid N''$	$N \xrightarrow{\epsilon_0/\epsilon} N'$ and $N \xrightarrow{\epsilon_1/\epsilon} N''$

The *semantics* of p is the greedy semantics of its associated transducer: $\mathcal{G}[\![p]\!] = \mathcal{G}[\![\mathcal{T}_p]\!](N_0)$ where N_0 is a designated start nonterminal. (By convention, this is `main`.)

3.2 Standard Kleenex

We extend the syntax of right-hand sides in Kleenex productions with arbitrary concatenations of the form and $N'N''$ and slightly simplify the remaining rules as follows:

$$t ::= \varepsilon \mid N \mid a \mid \text{"}b\text{"} \mid N_0 \,|\, N_1 \mid N'N''$$

Let p be such a *Standard Kleenex* program. Its *dependency graph* $G_p = (\mathcal{N}, D)$ consists of its nonterminals $\mathcal{N}$ and the *dependencies* $D = \{N \to N' \mid N'$ occurs in the definition of N in $p\}$. Define the *strict dependencies* $D_s = \{N \to N' \mid (N{:=}N'N'') \in p\}$.

Definition 8 (Well-formedness)**.** A Standard Kleenex program p is *well-formed* if no strong component of G_p contains a strict dependency.

Well-formedness ensures that the underlying grammar is non-self-embedding [10], and thus its input language is regular.

Definition 9 (Kleenex syntax and semantics)**.** Let p be a well-formed Kleenex program with nonterminals $\mathcal{N}$. Define the transitions $E \subseteq \mathcal{N}^* \times \Sigma[\epsilon_0, \epsilon_1, \epsilon] \times \Gamma[\epsilon] \times \mathcal{N}^*$ as follows:

For rule d	add these transitions for all $X \in \mathcal{N}^*$ to E
$N{:=}\,\varepsilon$	$NX \xrightarrow{\epsilon/\epsilon} X$
$N{:=}\,N'$	$NX \xrightarrow{\epsilon/\epsilon} N'X$
$N{:=}\,a$	$NX \xrightarrow{a/\epsilon} X$
$N{:=}$ "b"	$NX \xrightarrow{\epsilon/b} X$
$N{:=}\,N'\,N''$	$NX \xrightarrow{\epsilon/\epsilon} N'N''X$
$N{:=}\,N' \mid N''$	$NX \xrightarrow{\epsilon_0/\epsilon} N'X$ and $NX \xrightarrow{\epsilon_1/\epsilon} N''X$

Let $\mathrm{Reach}(N) = \{\vec{N}_k \mid N \xrightarrow{\cdot/\cdot} \ldots \xrightarrow{\cdot/\cdot} \vec{N}_k\}$ be the nonterminal sequences reachable from N along transitions in E. The *transducer* $\mathcal{T}_p$ *associated with* p is $(\Sigma, \Gamma, R, N, \varepsilon, E|_R)$ where $R = \mathrm{Reach}(N)$ for designated start symbol N and $E|_R$ is E restricted to R. The *(greedy) semantics* of p is the greedy semantics of $\mathcal{T}_p$: $\mathcal{G}[\![p]\!] = \mathcal{G}[\![\mathcal{T}_p]\!]$.

The following proposition justifies calling $\mathcal{T}_p$ a transducer.

Proposition 4. *Let p be a well-formed Standard Kleenex program, with $\mathcal{T}_p$ as defined above. Then R is finite, and $\mathcal{T}_p$ is a transducer, that is normalized FST.*

Proof. (Sketch) $\mathrm{Reach}(N)$ consists of all the nonterminal suffixes of sentential forms of left-most derivations of p considered as a context-free grammar. In well-formed Kleenex programs, their maximum length is bounded by $|\mathcal{N}|$. It is easy to check that every state in R is either a resting, skip, choice or final state. □

Observe that the transducer associated with a Kleenex program can be exponentially bigger than the program itself.

Since a transducer has a straightforward representation in Core Kleenex, the construction of $\mathcal{T}_p$ provides a translation of a well-formed Standard Kleenex program into Core Kleenex. For example, the Kleenex program on the left translates into the Core Kleenex program on the right:

$$\begin{array}{rcl} M & := & M' \,|\, N \\ M' & := & N N_a \\ N_a & := & a \\ N & := & N' \,|\, N_\varepsilon \\ N' & := & N_b N \\ N_b & := & b \\ N_\varepsilon & := & \varepsilon \end{array} \quad\Longrightarrow\quad \begin{array}{rcl} M & := & N' \,|\, N \\ M' & := & N' \,|\, N_a \\ N_a & := & aN_\varepsilon \\ N' & := & bM' \\ N & := & N' \,|\, N_\varepsilon \\ N_\varepsilon & := & \varepsilon \end{array}$$

3.3 The Full Surface Language

The full surface syntax of *Kleenex* is obtained by extending Standard Kleenex with the following term-level constructors, none of which increase the expressive power:

$$t ::= \ldots \mid \text{"}v\text{"} \mid /e/ \mid \text{\textasciitilde}t \mid t_0 t_1 \mid t_0 \,|\, t_1 \mid t\texttt{*} \mid t\texttt{+} \mid t\texttt{?}$$
$$\mid t\{n\} \mid t\{n,\} \mid t\{,m\} \mid t\{n,m\}$$

where $v \in \Gamma^*$, $n, m \in \mathbb{N}$, and e is a *regular expression*. The terms $t_0 t_1$ and $t_0 \,|\, t_1$ desugar into $N_0 N_1$ and $N_0 \,|\, N_1$, respectively, with additional productions $N_0 := t_0$ and $N_1 := t_1$ for new nonterminals N_0, N_1. The term "v" is shorthand for a sequence of outputs.

Regular expressions are special versions of Kleenex terms without nonterminals. They desugar to terms that output the matched input string, i.e. /e/ desugars by adding an output symbol "a" after every input symbol a in e. For example, the regular expression /a*|b{n,m}|c?/ becomes $(a"a")$*|$(b"b")\{n,m\}$|$(c"c")$?, which can then be further desugared.

A *suppressed* subterm ~t desugars into t with all output symbols removed, including any that might have been added in t by the above construction. For example, ~("b"/a/) desugars into ~("b" a "a"), which further desugars into a.

The operators ·*, ·+ and ·? desugar to their usual meaning as regular operators, as do the repetition operators $\cdot\{n\}$, $\cdot\{n,\}$, $\cdot\{,m\}$, and $\cdot\{n,m\}$. Note that they all desugar into their *greedy* variants where matching a subexpression is preferred over skipping it. For example:

$$M \;:=\; (a\,\text{"}b\text{"})+ \quad\Longrightarrow\quad \begin{array}{lcl} M & := & (a\,\text{"}b\text{"})N' \\ N' & := & a\,\text{"}b\text{"}N' \mid \varepsilon \end{array}$$

Lazy variants can be encoded by making ε the left rather than the right choice of an alternative.

3.4 Register Update Actions

By viewing Γ as an alphabet of *effects*, we can extend the expressivity of Kleenex beyond rational functions [13]. Let X be a computable set, and assume that there is an effective partial action $\Gamma \times X \to X$. It is simple to define a deterministic machine implementing the function $\Gamma^* \times X \to X$ by successively applying a list of actions to some starting state X. Any Kleenex program then denotes a function $\Sigma^* \times X \to X$ by composing its greedy semantics with such a machine. If we can implement the pure transducer part in a streaming fashion, then a state X can be maintained on-the-fly by interpreting output actions as soon as they become available.

Let $X = (\Gamma^*)^+ \times (\Gamma^*)^n$ for some n, representing a non-empty stack of output strings and n string registers. The transducer output alphabet is extended to $\Gamma[\mathsf{push}, \mathsf{pop}_0, ..., \mathsf{pop}_n, \mathsf{write}_0, ..., \mathsf{write}_n]$, with actions defined by

$$\begin{aligned} (t\vec{w}, v_0, ..., v_n) \cdot a &= ((ta)\vec{w}, v_0, ..., v_n) \qquad (a \in \Gamma) \\ (\vec{w}, v_0, ..., v_n) \cdot \mathsf{push} &= ((\varepsilon)\vec{w}, v_0, ..., v_n) \\ (t\vec{w}, v_0, ..., v_i, ..., v_n) \cdot \mathsf{pop}_i &= (\vec{w}, v_0, ..., t, ..., v_n) \quad (|\vec{w}| > 0) \\ (t\vec{w}, v_0, ..., v_n) \cdot \mathsf{write}_i &= ((tv_i)\vec{w}, v_0, ..., v_n) \end{aligned}$$

The bottom stack element can only be appended to and models a designated output register—popping it is undefined. The stack and the variables can be used to perform complex string interpolation. To access the extended actions, we extend the surface language:

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

where R ranges over *register names* standing for indices.

The term R @ t desugars to "push" t "pop$_R$", and the term !R desugars to "write$_R$". The term [R <- $x_1 ... x_m$] desugars to "push"$t'_1 ... t'_m$"pop$_R$", where $t'_i = \mathsf{write}_{R_i}$ if $x_i = R_i$, and $t'_i = x_i$ otherwise. Finally, [R += $\vec{x}$] desugars to [R <- $R\,\vec{x}$].

Thus all streaming string transducers (see Section 5) can be coded. 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/
```

where the first line above desugars to

```
main := "push" line "pop_a" "push" line "pop_b"
        "write_b" "write_a"
```

4. Streaming Simulation

As we have seen, every Kleenex program has an associated transducer, which can be split into oracle and action machines. The action machine is a straightforwardly implemented deterministic FST. The oracle machine is nondeterministic, however: The key challenge is how to (deterministically) find and output the lexicographically least path that accepts a given input string. In this section we develop an efficient oracle machine simulation algorithm that inputs a stream of symbols and streams the output bits almost as early as possible during input processing.

4.1 Path Trees

Given an oracle machine $\mathcal{T}^{\mathsf{C}}$ as in Definition 4, consider input s such that $q^- \xrightarrow{u/v}_{\mathsf{min}} q^f$ where $|u| = s$. Recall that $q \xrightarrow{u/v}_{\mathsf{min}} q'$ uniquely identifies a path from q to q' in $\mathcal{T}^{\mathsf{C}}$, which is furthermore asserted to be the lexicographically minimal amongst all nonproblematic paths from q to q'.

Proposition 5 (Path decomposition)**.** *Assume $q^- \xrightarrow{u/v}_{\mathsf{min}} q^f$. For every prefix s' of $|u|$ there exist unique u', v', u'', v'', q' such that $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q' \xrightarrow{u''/v''}_{\mathsf{min}} q^f$, q' is a resting state, $|u'| = s'$, $u'u'' = u$ and $v'v'' = v$.*

Proof. Let u' be the longest prefix of u such that $|u'| = s'$ and let $q^- \xrightarrow{u'/v'}_{\mathsf{np}} q'$ be the path from q determined by u'. (Such a prefix must exist.) Claim: This is the q' in the proposition.

1. q' is a resting state. If it were not, we could transition on ϵ, ϵ_0 or ϵ_1 resulting in a longer prefix w with $|w| = s'$.
2. $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q'$ and $q' \xrightarrow{u''/v''}_{\mathsf{min}} q^f$. If any of these subpaths were not lexicographically minimal, we could replace it with one that is lexicographically less, resulting in a path from q^- to q^f that is lexicographically less than $q^- \xrightarrow{u/v}_{\mathsf{np}} q^f$, contradicting our assumption $q^- \xrightarrow{u/v}_{\mathsf{min}} q^f$. □

After reading input prefix s' we need to find the above $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q'$ where $|u'| = s'$. Since we do not know the remaining input yet, however, we maintain *all* paths $q^- \xrightarrow{u'/v'}_{\mathsf{min}} q'$ for any resting state q' such that $|u'| = s'$.

Definition 10 (Path tree)**.** Let $\mathcal{T}^{\mathsf{C}}$ be given. Its *path tree* $\mathrm{P}(s)$ for s is the set of paths $\{q^- \xrightarrow{u/v}_{\mathsf{min}} q' \mid |u| = s\}$.

Consider a transducer as a directed labeled graph where the nodes are transducer states indexed by the strings reaching them, $\{q_s \mid \exists u, v.\, q^- \xrightarrow{u/v} q \wedge |u| = s\}$, and the edges are the corresponding transitions, $\{q_s \xrightarrow{a/b} q'_{s\bar{a}} \mid q \xrightarrow{a/b} q'\}$. It can be seen that $\mathrm{P}(s)$ is a subgraph that forms a non-full rooted edge-labeled binary tree. The *stem* of $\mathrm{P}(s)$ is the longest path in this tree from q^-_ε to some $q_{s'}$ for a prefix s' of s only involving nodes with at most one child. The *leaves* of $\mathrm{P}(s)$ are the states q such that q_s is reachable, in lexicographic order of the paths reaching them from q^-_ε.

Example 2. Recall the oracle machine for the decimal converter in the lower left of Figure 1. Its path tree for input a is shown in the upper left of Figure 2. The nodes are subscripted with the length of the input prefix rather the input prefix itself. Note that the leaf states are listed from top to bottom in lexicographic order of their paths reaching them. This means that the top state is the prime candidate for being q' in Proposition 5. If the remainder of the input is not accepted from it, though, the other leaf states take over in the given order.

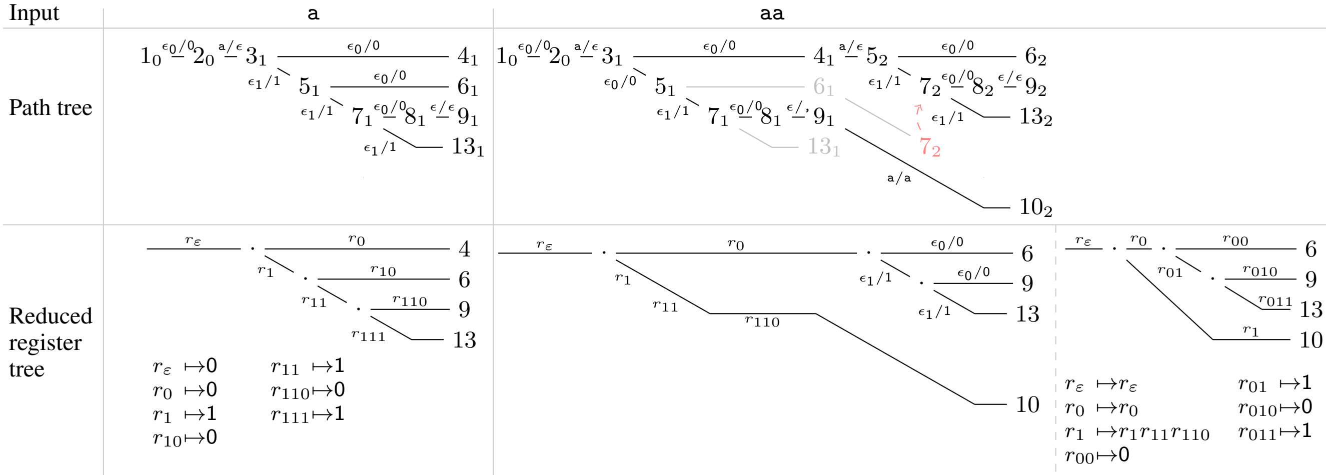

Figure 2: Path trees for the decimal conversion oracle in Figure 1. Above left: path tree reading a. Subscripts denote the number of input symbols read when the given state was visited. Above right: path tree reading aa. Failing paths are shown in gray. Below left: reduced register tree reading a, with register valuation. Below middle: extension of register tree after reading an additional a. Note mix of registers and bits, and that bottom branch is now labeled by a sequence of registers. Below right: the path tree and register *update* after reading aa. The registers r_1, r_{11}, and r_{110} are all on the same unary path and are concatenated.

4.2 Basic Simulation Algorithm

The basic streaming simulation algorithm works as follows:

Algorithm 1 (Basic streaming algorithm)**.** Let $s = a_1 \dots a_n \in \Sigma^*$ be the input string.

- **for** $i = 1$ **to** n **do**
  - **if** $\mathrm{P}(a_1...a_i) = \emptyset$ **then**
    - **terminate** with failure (input rejected)
  - **if** $\mathrm{stem}(\mathrm{P}(a_1...a_i))$ longer than $\mathrm{stem}(\mathrm{P}(a_1...a_{i-1}))$ **then**
    - emit the output bits on the stem extension
- **if** $\mathrm{P}(a_1...a_n)$ contains path to q^f **then**
  - if path tree contains at least one branch, emit output bits on path from highest binary ancestor to q^f
  - **terminate** with success (input accepted)
- **else**
  - **terminate** with failure (input rejected)

The critical step in the algorithm is incrementally computing the path tree for $s'a$ from the path tree for s'.

Algorithm 2 (Incremental path tree computation)**.** Let P be $\mathrm{P}(s')$ for some prefix s' of the input string, and let $[q_0, ..., q_n]$ be its leaves in lexicographic order of the paths reaching them. Upon reading a, incrementally compute $\mathrm{P}(s'a)$ as follows:

1: **for** $q = q_0$ **to** q_n **do**
2: compute $\mathrm{P}_q(a)$, the path tree of lexicographically least (u/v) paths with $\overline{u} = a$ from q to resting states, but excluding resting states that have been reached in a previous iteration
3: **if** $\mathrm{P}_q(a)$ is non-empty **then**
4: replace leaf node q in P by $\mathrm{P}_q(a)$
5: **else**
6: prune branch from lowest binary ancestor to leaf node q; if binary ancestor does not exist, then **terminate** with failure (input rejected)

Example 3. The upper right in Figure 2 shows P(aa) for the decimal converter. Observe how it arises from P(a) by extending leaf states 4 and 9, which have an a-transition, and building the ϵ-closure as a binary tree. It prunes branches either because they reach a state already reached by a lexicographical lower path (state 6) or because the leaf does not have transition on a (state 13). The algorithm outputs 0 after reading the first a since 0 is the sequence of output bits on the stem of the path tree. It does not output anything after reading the second a since $\mathrm{P}(aa)$ has the same stem as $\mathrm{P}(a)$.

Definition 11 (Optimal streaming)**.** Let f be a partial function from Σ^* to Γ^*, $s \in \Sigma^*$. Let $T(s) = \{f(ss') \mid s' \in \Sigma^* \wedge ss' \in \mathrm{dom} f\}$. The *output* $f^{\#}(s)$ *determined by* f *for* s is the longest common prefix of $T(s)$ if $T(s)$ is nonempty; otherwise it is undefined. The partial function $f^{\#}$ is called the *optimally streaming version* of f. An *optimally streaming algorithm* for f is an algorithm that implements $f^{\#}$: It emits output symbols as soon as they are semantically determined by the input prefix read so far.

Let transducer $\mathcal{T}$ be given. Write $\mathcal{L}[\![q]\!]$ for $\mathcal{L}[\![\mathcal{T}']\!]$ where $\mathcal{T}'$ is $\mathcal{T}$, but with q as initial state instead of q^-. A state q is *covered* by $\{q_1, \dots, q_k\}$ if $\mathcal{L}[\![q]\!] \subseteq \mathcal{L}[\![q_1]\!] \cup \dots \cup \mathcal{L}[\![q_k]\!]$. A path tree $\mathrm{P}(s)$ with lexicographically ordered leaves $[q_1, \dots, q_n]$ is *cover-free* if no q_i is covered by $\{q_1, \dots, q_{i-1}\}$. $\mathcal{T}$ is *cover-free* if $\mathrm{P}(s)$ is cover-free for all $s \in \Sigma^*$.

Theorem 1. *Let $\mathcal{T}$ be cover-free. Then Algorithm 1 with Algorithm 2 for incremental path tree recomputation is an optimally streaming algorithm for $\mathcal{G}[\![\mathcal{T}^{\mathsf{C}}]\!]$ that runs in time $O(mn)$, where $m = |\mathcal{T}^{\mathsf{C}}|$ and n is the length of the input string.*

Proof. (Sketch) Algorithm 2 can be implemented to run in time $O(m)$ since it visits each transition in $\mathcal{T}^{\mathsf{C}}$ at most once and pruning can be amortized: every deallocation of an edge can be charged to its allocation. Algorithm 1 invokes Algorithm 2 n times. Optimal streaming follows from a generalization of the proof of optimal streaming for regular expression parsing [32]. □

The algorithm can be made optimally streaming for all oracle transducers by also pruning leaf states that are covered by other leaf states in Step 6 of Algorithm 2. Coverage is PSPACE-complete, however. Eliding the coverage check does not seem to make much of a difference to the streaming behavior in practice.

5. Determinization

NFA simulation maintains a set of NFA states. This is the basis of compiling an NFA into a DFA: precompute and number the set of *all* NFA state sets reachable by *any* input from the initial NFA state, observing that there are only finitely many such sets. In the transducer simulation in Section 4 path trees play the role of NFA state sets. The corresponding determinization idea does not work for transducers, however: $\{\mathrm{P}(s) \mid s \in \Sigma^*\}$ is in general infinite. For example, for the oracle machine in Figure 1, the trees $\mathrm{P}(\mathtt{a}^n)$ all have the same stem, but contain paths with bit strings of length proportional to n. This is inherently so. A single-valued transducer can be transformed effectively [12, 66] into a form of deterministic finite-state transducer if its relational semantics is *subsequential* [13, 53], but nondeterministic finite state transducers in general are properly more expressive than their deterministic counterparts. We can factor a path tree into its underlying full binary tree and the labels associated with the edges, though. Since there are only finitely many different such trees, we can achieve determinization to transducers with registers storing the potentially unbounded label data.

Definition 12 (Streaming String Transducer [4])**.** A deterministic *streaming string transducer* (SST) over alphabets Σ, Γ is a tuple $\mathcal{S} = (X, Q, q^-, F, \delta^1, \delta^2)$ where

- X is a finite set of *register variables*;
- Q is is a finite set of *states*;
- F is a partial function $Q \to (\Gamma \cup X)^*$ mapping each *final state* $q \in \mathrm{dom}(F)$ to a word $F(q) \in (\Gamma \cup X)^*$ 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 $Q \times \Sigma \to (X \to (\Gamma \cup X)^*)$ such that for each $q \in Q$, $a \in \Sigma$ and $x \in X$, there is at most one occurrence of x in the multiset of strings $\{\delta^2(q,a)(y) \mid y \in X\}$.

A *configuration* of an SST $\mathcal{S} = (X, Q, q^-, F, \delta^1, \delta^2)$ is a pair (q, ρ) where $q \in Q$ is a state, and $\rho : X \to \Gamma^*$ is a *valuation*. A valuation extends to a monoid homomorphism $\widehat{\rho} : (X \cup \Gamma)^* \to \Gamma^*$ by setting $\rho(x) = x$ for $x \in \Gamma$. The initial configuration is (q^-, ρ^-) where $\rho^-(x) = \epsilon$ for all $x \in X$.

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

Every SST $\mathcal{S}$ denotes a partial function $\mathcal{F}[\![\mathcal{S}]\!] : \Sigma^* \to \Gamma^*$ where for any $u \in \Sigma^*$ such that $\delta^*((q^-, \rho^-), u) = (q', \rho')$, we define

$$\mathcal{F}[\![\mathcal{S}]\!](u) = \begin{cases} \widehat{\rho'}(F(q')) & \text{if } q' \in \mathrm{dom}(F) \\ \text{undefined} & \text{otherwise} \end{cases}$$

In the following, let $X = \{r_p \mid p \in \mathbf{2}^*\}$ be a set of registers.

Definition 13 (Reduced register tree)**.** Let P be a path tree. Its *reduced register tree* $\mathcal{R}(\mathrm{P})$ is a pair $(R_\mathrm{P}, \rho_\mathrm{P})$ where ρ_P is a valuation $X \to \mathbf{2}^*$ and R_P is a full binary tree with state-labeled leaves, obtained from P by first contracting all unary branches and concatenating edge labels; then replacing each edge label (u/v) by a single register symbol r_p, where p denotes the unique path from the root to the edge destination node, and setting $\rho_\mathrm{P}(r_p) = v$.

The set $\{R_{\mathrm{P}(s)} \mid s \in \Sigma^*\}$ is finite: it is bounded by the number of full binary trees with up to $|Q|$ leaves times the number of possible permutations of the leaves.

Let R be R_P and $a \in \Sigma$ a symbol, and apply Algorithm 2 to R. The result is a non-full binary tree with edges labeled either by a register or by a (u/v) pair. By reducing the tree again and treating registers as output labels, we get a pair $(R_a, \kappa_{R,a})$ where $\kappa_{R,a} : X \to (\mathbf{2} \cup X)^*$ is a register update.

Example 4. Consider the bottom left tree in Figure 2. This is the reduced register tree obtained from the path tree above it. The evaluation map ρ can be seen below it, where register subscripts denote their position in the register tree. In the middle is the result of extending the register tree using Algorithm 2. Reducing this again yields the tree on the right. The update map κ is shown below it—note that the range of this map is mixed register/bit sequences.

Proposition 6. *Let $\mathcal{T}^\mathsf{C}$ be given, and let $\mathrm{P} = \mathrm{P}(s)$, $\mathrm{P}' = \mathrm{P}(sa)$, $(R, \rho) = \mathcal{R}(\mathrm{P})$ and $(R', \rho') = \mathcal{R}(\mathrm{P}')$ for some s and a. Then $R' = R_a$ and $\rho' = \widehat{\rho} \circ \kappa_{R,a}$.*

Theorem 2. *Let $\mathcal{T}^\mathsf{C}$ be an oracle machine of size m. There is an SST $\mathcal{S}$ with $O(2^{m \log m})$ states such that $\mathcal{F}[\![\mathcal{S}]\!] = \mathcal{G}[\![\mathcal{T}]\!]$.*

Proof. Let $Q_\mathcal{S} = \{R_{\mathrm{P}(s)} \mid s \in \Sigma^*\} \cup \{R_0\}$ and $q^-_\mathcal{S} = R_0$, where R_0 is the single-leaf binary tree with leaf $q^-_\mathcal{T}$. The set of registers $X_\mathcal{S}$ is the finite subset of register variables occurring in $Q_\mathcal{S}$. The transition maps are given by $\delta^1_\mathcal{S}(R, a) = R_a$ and $\delta^2_\mathcal{S}(R, a) = \kappa_{R,a}$. For any $R \in Q_\mathcal{S} - \{R_0\}$, define the final output $F_\mathcal{S}(R)$ to be the sequence of registers on the path from the root to the final state $q^f_\mathcal{T}$ in R if R contains it as a leaf; otherwise let $F_\mathcal{S}(R)$ be undefined. Let $F_\mathcal{S}(R_0) = \overline{v}$ if $q^-_\mathcal{T} \xrightarrow{\epsilon/v}_{\mathsf{min}} q^f$ for some v; otherwise let $F_\mathcal{S}(R_0)$ be undefined.

Correctness follows by showing $\delta^*((R_0, \rho^-), u) = \mathcal{R}(\mathrm{P}(u))$ for all $u \in \Sigma^+$. We prove this by induction, applying Proposition 6 in each step. For the case $u = \varepsilon$ correctness follows by the definition of $F_\mathcal{S}(R_0)$.

The upper bound follows from the fact that there are at most $C_{k-1}(k-1)! = O(2^{m \log m})$ full binary trees with k pairwise distinct leaves where k is the number of resting states in $\mathcal{T}^\mathsf{C}$ and C_{k-1} is the $(k-1)$-st *Catalan* number. ☐

Example 5. The oracle machine in Figure 1 yields the SST in Figure 3. The states 1 and 2 are identified by the left and right reduced trees, respectively, in the bottom of Figure 2.

Corollary 1. *The SST $\mathcal{S}$ for $\mathcal{T}^\mathsf{C}$ can be implemented to execute in time $O(mn)$ where $m = |\mathcal{T}^\mathsf{C}|$.*

Proof. (Sketch) Use a data structure for imperatively extending a string register, $r := rs$, in amortized time $O(n)$ where n is the size of s, independent of the size of the string stored in r. The result then follows from the fact that the steps in Algorithm 2 can be implemented in the same amortized time. ☐

In practice, the compiled version of the SST is much more efficient—roughly one to two orders of magnitude faster—than streaming simulation since it compiles away the interpretive overhead of explicitly managing the binary trees underlying path trees and employs machine word-level parallelism by operating on bit strings in fewer registers rather than many edges each labeled by at most one bit.

6. Implementation and Benchmarks

Our implementation[3] compiles the action machine and the oracle SST to machine code via C. We have implemented several optimizations which are orthogonal to the underlying principles behind our compilation from Kleenex via transducers to SSTs:

Inlining of output actions The action machine and the oracle SST need to be composed. We can do this at runtime by piping the SST output to the action machine, or we can apply a form of deforestation [70] to inline the output actions directly into the SST. This is straightforward since the machines are deterministic.

[3] Source code and benchmarks available at `http://kleenexlang.org/`

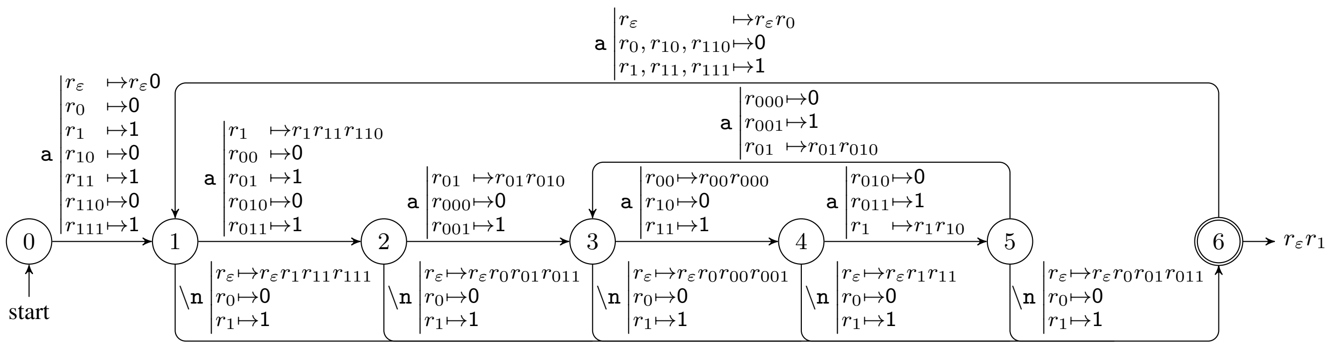

Figure 3: SST constructed from the oracle machine in Figure 1.

Constant propagation The SSTs generated by the construction underlying Theorem 2 typically contain many constant-valued registers (e.g. most registers in Figure 3 are constant). We eliminate these using constant propagation: compute reaching definitions by solving a set of data-flow constraints.

Symbolic representation A more succinct SST representation is obtained by using a symbolic representation of transitions where input symbols are replaced by *predicates* and output symbols by *terms* indexed by input symbols. This is a straightforward extension of similar representations for automata [72] and transducers [66–69]. Our implementation uses simple predicates in the form of byte ranges, and simple output terms represented by byte-indexed lookup tables. We refer the reader to the cited literature for the technical details of symbolic transducers.

Finite lookahead Symbolic FSTs with bounded lookahead have been shown to reduce the state space when representing string encoders [22, 67, 69]. We have implemented a form of finite lookahead in our SST representation. Opportunities for lookahead is detected by the compiler, and arise in the case where the program contains a string constant with length above one. In this case a lookahead transition is used to check once and for all if the string constant is matched by the input instead of creating an SST state for each symbol. This may in some cases reduce the size of the generated code since we avoid tabulating all states of the whole program for every prefix of the string constant.

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

RE2, Google's regular expression C++ library [62].
RE2J, a recent re-implementation of RE2 in Java [63].
GNU `AWK` **and GNU** `sed`**,** programming languages and tools for text processing and extraction [60].
Oniglib, a regular expression library written in C++ with support for different character encodings [38].
Ragel, a finite state machine compiler with multiple language backends [65].

In addition, we implemented test programs using the standard regular expression libraries in the scripting languages Perl [71], Python [41], and Tcl [73].

The benchmark suite, Kleenex programs, and version numbers of libraries used can be found at `http://kleenexlang.org`.

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 Section 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. 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 times out (after 30 seconds). As we fully determinize the transducers, the resulting C code can explode in some cases. The two worst-case exponential blow-ups in generating transducers from Kleenex and then generating SSTs implemented in C code from transducers are *inherent*, though, and as such can be considered a *feature* of Kleenex: tools based on finite machines with no or limited nondeterminism support such as Ragel would require *hand-coding* a potentially huge machine that Kleenex generates *automatically*.[4]

6.1 Baseline

The following two 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.

[4] We have found it excessively difficult to employ Ragel in some use cases with a natural nondeterministic specification.

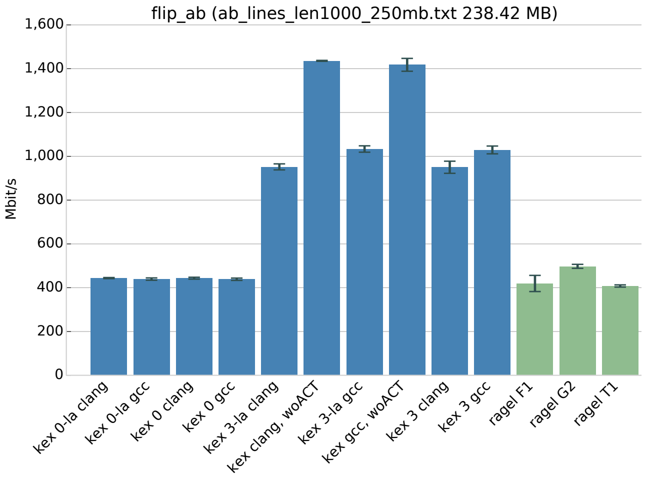

Figure 4: `flip_ab` run on lines with average length 1000.

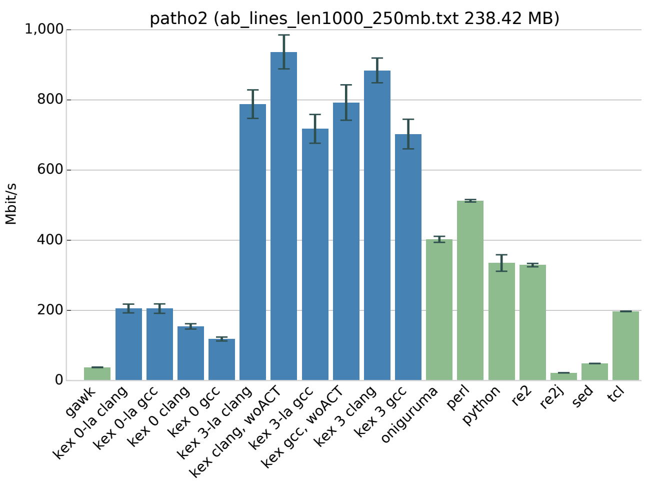

Figure 5: `patho2` run on lines with average length 1000.

The performance of the two implementations run on input with an average line length of 1000 characters is shown in Figure 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 Figure 5 shows. Due to the high degree of interleaving and the lack of keywords, in this program the lookahead optimization has reduced 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 for us 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]/
```

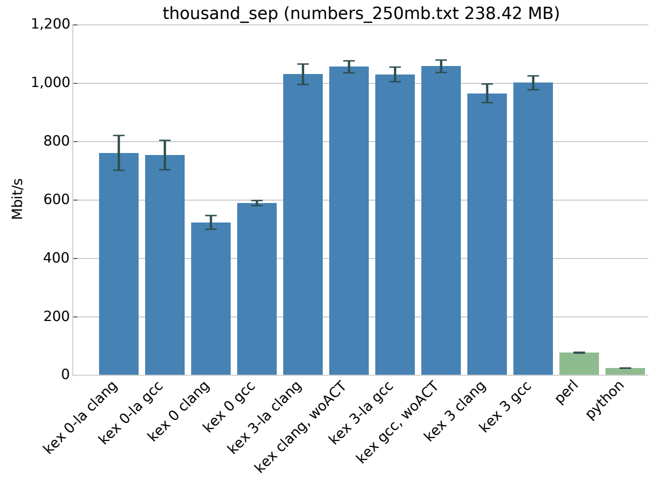

Figure 6: Inserting separators in random numbers of average length 1000.

We evaluated the Kleenex implementation along with two other implementations using Perl and Python. The performance can be seen in Figure 6. Both Perl and Python are significantly slower than all of the Kleenex implementations; the problem is tricky to solve with regular expressions unless one reads the input right-to-left.

IRC protocol handling The following Kleenex program parses the IRC protocol as specified in RFC 2812.[5] It follows roughly the output style described in part 2.3.1 of the RFC. Note that the Kleenex source code and the BNF grammar in the RFC are almost identical. Figure 7 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 )?
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}
```

CSV rewriting The program `csv_project3` deletes all columns but the 2nd and 5th from a CSV file:

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

Various specialized tools that can handle this transformation are included in Figure 8; GNU `cut` is a command that splits its input on certain characters, and GNU `AWK` has built-in support for this type of transformation.

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

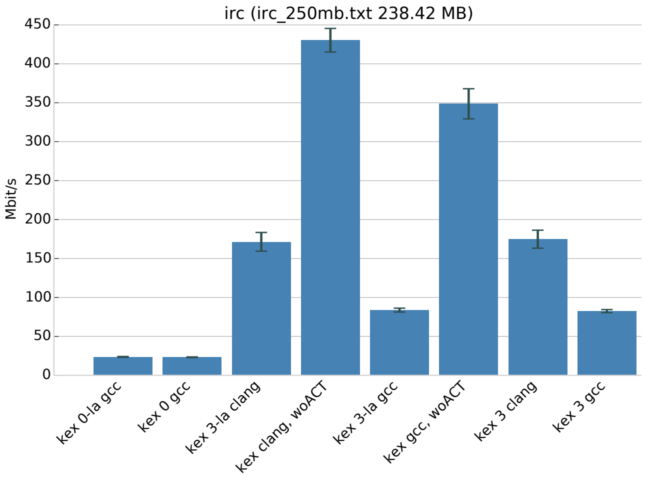

Figure 7: Throughput when parsing 250 MiB random IRC data.

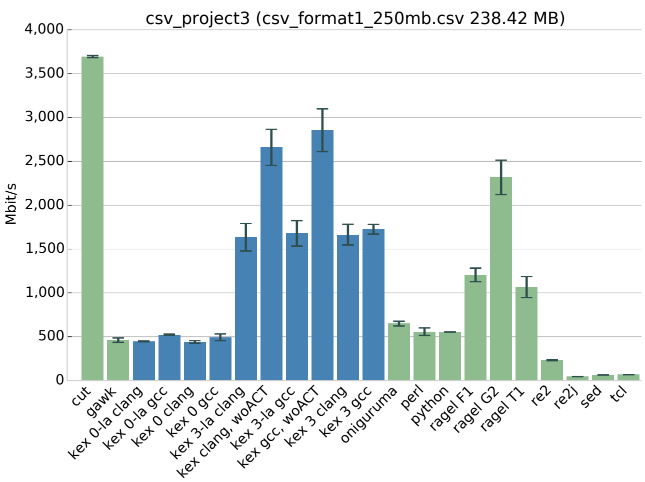

Figure 8: `csv_project3` reads in a CSV file with six columns and outputs columns two and five. "gawk" is GNU `AWK` that uses the native `AWK` way of splitting up lines. "cut" is a tool from GNU coreutils that splits up lines.

Apart from `cut`, which is very fast for its own use case, a 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 which is flushed routinely, whereas a programmer has to manually handle buffering when writing a Ragel program.

6.3 With or Without Action Separation

One can choose to use the machine resulting from fusing the oracle and action machines when compiling Kleenex. Doing so results in only one process performing both disambiguation and outputting, which in some cases is faster and in other cases slower. Figures 8, 9, and 11 illustrate both situations. It depends on the structure of the problem whether it pays off to split up the work into two processes; if all the work happens in the oracle machine and the action machine does nearly nothing, then the added overhead incurred by the process context switches becomes noticeable. On the other hand, in cases where both machines perform much work, the fact that two CPU cores can be utilized in parallel speeds up execution. This is more likely once Kleenex has support for actions that can perform arbitrary computations, e.g. in the form of embedded C code.

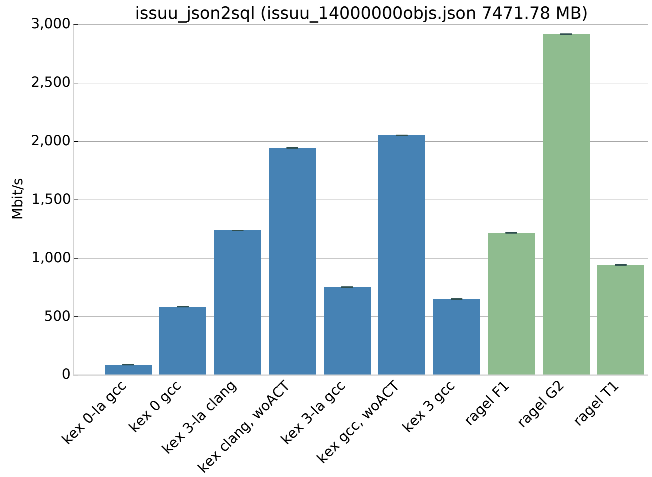

Figure 9: The speeds of transforming JSON objects to SQL INSERT statements using Ragel and Kleenex.

7. Use Cases

We briefly touch upon various use cases—natural application scenarios—for Kleenex.

JSON logs to SQL We have implemented a Kleenex program that transforms a JSON log file into an SQL insert statement. The program works on the logs provided by Issuu.[6]

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

Apache CLF to JSON The Kleenex program below rewrites Apache CLF[7] 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\":\"" /-/ "\""
authuser := "\"authuser\":\"" /[^ \n]+/ "\""
timestamp := "\"date\":\"" ~/\[/ /[^\n\]]+/ ~/]/ "\""
request := "\"request\":" quotedString
code := "\"status\":\"" integer "\""
bytes := "\"size\":\"" (integer | /-/) "\""
referer := "\"url\":" quotedString
useragent := "\"agent\":" quotedString
sep := "," ~/[\t ]+/
quotedString := /"([^"\n]|\\")*"/
integer := /[0-9]+/
ip := integer (/\./ integer){3}
```

This is a re-implementation of a Ragel program.[8] Figure 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 [30], the program `iso_datetime_to_json` converts date and time

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

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

[8] `https://engineering.emcien.com/2013/04/5-building-tokenizers-with-ragel`

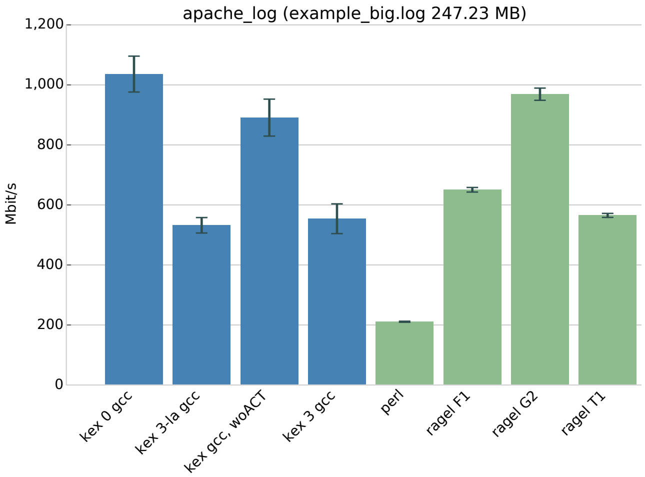

Figure 10: Speed of the conversion from the Apache Common Log Format to JSON.

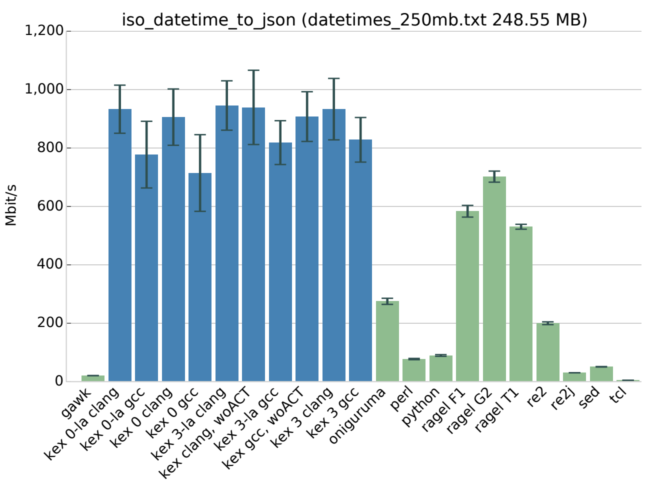

Figure 11: The performance of the conversion of ISO time stamps into JSON format.

stamps in an ISO standard format to a JSON object. Figure 11 shows the performance.

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 <- "" ]
```

Syntax highlighting Kleenex can be used to write syntax highlighters; in fact, the Kleenex syntax in this paper was highlighted using a Kleenex program.

8. Discussion

We discuss related and future work by building Kleenex conceptually up from regular expression matching via regular expressions as types for bit-coded parsing to transducers and eventually grammars with embedded actions.

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

For *acceptance testing*, the subject of *automata theory* where only a single bit is output, NFA-simulation and DFA-construction are classical techniques. Bille and Thorup [14] improve on Myers' [46] log-factor improved classical NFA-simulation for regular expressions, based on tabling. They design an $O(kn)$ algorithm [15] with word-level parallelism, where $k \leq m$ is the 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 [34], an input is not only classified as accepting or not, but a substring is returned for each sub-RE of interest. Subgroup matching exposes ambiguity in the RE. Subgroup matching is often implemented by backtracking over alternatives, which implements *greedy* disambiguation.[9] Backtracking may result in exponential-time worst case behavior, however, even in the absence of inherently hard matching with backreferences [1]. Considerable human effort is usually expended to engineer REs used in practice to perform well anyway. More recently, REs designed to force exponential run-time behavior are used in algorithmic attacks, though [52, 56]. Some subgroup matching libraries have guaranteed worst-case linear-time performance based on automata-theoretic techniques, notably Google's RE2 [62]. Intel's Hyperscan [61] is also described as employing automata-theoretic techniques. A key point of Kleenex is implementing the natural backtracking semantics without actually performing backtracking and without requiring storage of the input.

Myers, Oliva and Guimaraes [44] and Okui, Suzuki [50] describe a $O(mn)$, respectively $O(m^2n)$ *POSIX*-disambiguated matching algorithms. Sulzmann and Lu [57] use Brzozowski [20] and Antimirov derivatives [11] for Perl-style subgroup matching for greedy and POSIX disambiguation. Borsotti, Breveglieri, Reghizzi, and Morzenti [16, 17] have devised a Berry-Sethi based parser generator that can be configured for greedy or POSIX disambiguation.

Regular expression parsing. Full RE *parsing*, also called RE matching [29], generalizes subgroup matching to return a full parse tree. The set of parses are exactly the elements of a regular expression read as a *type* [29, 35]: 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 *(McNaughton-Yamada-)Thompson NFA* [42, 64] 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 [31, 33]. A Thompson NFA equipped with 0, 1 outputs [31] is a certain kind of oracle machine. The bit-code it generates can also be computed directly from the RE underlying the Thompson automaton [35, 49]. The *greedy RE parsing problem* produces the lexicographically least bit-code for a string matching a given RE. Kearns [37], Frisch and Cardelli [29] 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 [31] and an optimally streaming [32] greedy regular expression parsing algorithm. The algorithm works for all NFAs, indeed transducers, not just Thompson NFAs.

[9] Committing to the left alternative before checking that the remainder of the input is accepted is the essence of *parsing expression grammars* [28].

Sulzman and Lu [58] remark that POSIX is notoriously difficult to implement correctly and show how to use Brzozowski derivatives [20] for POSIX RE parsing.

Regular expression implementation optimizations. There are specialized RE matching tools and techniques too numerous to review comprehensively. We mention a few employing automaton optimization techniques potentially applicable to Kleenex, but presently unexplored. Yang, Manadhata, Horne, Rao, Ganapathy [75] 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 [54] implement NFAs directly on an FPGA, essentially performing NFA-simulation in parallel; it outperforms GNU `grep`. Brodie, Taylor, Cytron [18] 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 [55]. Navarro and Raffinot [48] show how to code DFAs compactly for efficient simulation.

Finite state transducers. From RE parsing it is a surprisingly short distance to the implementation of arbitrary nondeterministic finite state transducers (FSTs) [13, 43]. 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 noted, 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 [36], but may be inherently quadratically bigger than automata, even for DFAs [24, Theorem 23]. This is why Kleenex employs well-formed context-free grammars, which are much more compact than regular expressions.

Streaming string transducers. We have shown in this paper that the greedy semantics of arbitrary FSTs can be compiled to a *subclass* of streaming string transducers (SSTs). SSTs extensionally correspond to regular transductions, functions implementable by 2-way deterministic finite-state transducers [4], MSO-definable string transductions [25] and a combinator language analogous to regular expressions [8]. The implementation techniques used in Kleenex appear to be directly applicable to all SSTs, not just the ones corresponding to FSTs.

DReX [9] is a combinatory functional language for expressing all SST-definable transductions. Kleenex without register operations is expressively more restrictive; with copy-less register operations it appears to compactly code exactly the nondeterministic SSTs and thus SSTs. Programs in DReX must be unambiguous by construction while programs in Kleenex may be nondeterministic and ambiguous, which is greedily disambiguated.

Symbolic transducers. Veanes, Molnar, Mytkowics [69] employ symbolic transducers [23, 68] in the implementation of the Microsoft Research languages BEK[10] and BEX[11] for multicore execution. These techniques can be thought of as synthesizing code that implements the transition function of a finite state machine not only efficiently, but also compactly. Tabling in code form (switch statement) or data form (lookup in array) is the standard implementation technique for the transition function. It is efficient when applicable, but not compact enough for large alphabets and multistrided processing. Kleenex employs basic symbolic transition. Compact coding of multistrided transitions is likely to be crucial for exploiting word-level parallelism—processing 64 bits at a time—in practice.

[10] `http://research.microsoft.com/en-us/projects/bek`

[11] `http://research.microsoft.com/en-us/projects/bex`

Parallel transducer processing. Allender and Mertz [3] show that the functions computable by cost register automata [7], which generalize the string monoid used in SSTs to admit arbitrary monoids and more general algebraic structures, are in NC and thus inherently parallelizable. This appears to be achievable by performing relational FST-composition by matrix multiplication on the matrix representation of FSTs [13], which can be performed by parallel reduction. This requires in principle running an FST from all states, not just the input state, on input string fragments. Mytkowicz, Musuvathi, Schulte [47] observe that there is often a small set of cut states sufficient to run each FST. This promises to be an interesting parallel harness for a suitably adapted Kleenex implementation running on fragments of very large inputs.

Syntax-directed translation schemes. A Kleenex program is an example of a *syntax-directed translation scheme (SDTS)* or a domain-specific stream processing language such as PADS [26, 27] and Ziria [55]. In these the underlying grammar is typically deterministic modulo short lookahead so that semantic actions can be executed immediately when encountered during parsing.

Kleenex is restricted to non-self-embedding grammars to avoid the matrix-multiplication lower bound on general context-free parsing [40]; it supports full nondeterminism without lookahead restriction, though. A key contribution of Kleenex is that semantic actions are scheduled no earlier than semantically permissible and no later than necessary.

9. Conclusions

We have presented Kleenex, a convenient language for specifying nondeterministic finite state transducers, and its compilation to machine code implementing streaming string transducers.

Kleenex is comparatively expressive and performs consistently well. For complex regular expressions with nontrivial amounts of output it is almost always better than industrial-strength text processing tools such as RE2, Ragel, `AWK`, `sed` and RE-libraries of Perl, Python and Tcl in the evaluated use cases.

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, specifically visibly push-down transducers [51, 59], restricted versions of backreferences and approximate regular expression matching[45, 74];
- known, but so far unexplored optimizations, such as multistriding, automata minimization and symbolic representation, hybrid FST-simulation/SST-construction;
- massively parallel (log-depth, linear work) processing.

Acknowledgments

This work has been partially supported by The Danish Council for Independent Research under Project 11-106278, "Kleene Meets Church: Regular Expressions and Types"; see `http://diku.dk/kmc`. We would like to thank Alexandra Silva and Nate Foster for their critical questions and comments and the anonymous referees for their detailed reviews, which have given rise to numerous changes in the final version. We thank Issuu for releasing their data set to the research community. The work by the Jobindex authors was performed while being Master's students at DIKU.

The order of authors is insignificant; please list all authors—or none—when citing this paper.

References

[1] A. V. Aho. Algorithms for finding patterns in strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume Algorithms and Complexity (A), pages 255–300. Elsevier and MIT Press, 1990. ISBN 0-444-88071-2 and 0-262-22038-5.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006.

[3] E. Allender and I. Mertz. Complexity of regular functions. In *Proc. LATA*, 2015.

[4] R. Alur and P. Černỳ. Expressiveness of streaming string transducers. In *Proc. Foundations of Software Technology and Teoretical Computer Science (FSTTCS)*, 2010.

[5] R. Alur and P. Černỳ. Streaming transducers for algorithmic verification of single-pass list-processing programs. *ACM SIGPLAN Notices*, 46 (1):599–610, 2011.

[6] R. Alur and J. Deshmukh. Nondeterministic streaming string transducers. *Automata, Languages and Programming*, 2011.

[7] R. Alur, L. D'Antoni, J. Deshmukh, M. Raghothaman, and Y. Yuan. Regular functions and cost register automata. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 13–22. IEEE Computer Society, 2013.

[8] R. Alur, A. Freilich, and M. Raghothaman. Regular combinators for string transformations. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 9:1–9:10, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2886-9.

[9] R. Alur, L. D'Antoni, and M. Raghothaman. DReX: A declarative language for efficiently evaluating regular string transformations. In *Proc. 42nd ACM Symposium on Principles of Programming Languages (POPL)*, 2015.

[10] M. Anselmo, D. Giammarresi, and S. Varricchio. Finite automata and non-self-embedding grammars. In *Implementation and Application of Automata*, pages 47–56. Springer, 2003.

[11] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996. ISSN 0304-3975.

[12] M.-P. Béal and O. Carton. Determinization of transducers over finite and infinite words. *Theoretical Computer Science*, 289(1):225–251, Oct. 2002. ISSN 03043975.

[13] J. Berstel. *Transductions and Context-Free Languages*. Teubner, 1979.

[14] P. Bille and M. Thorup. Faster regular expression matching. In *Proc. 36th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 171–182, July 2009.

[15] P. Bille and M. Thorup. Regular expression matching with multi-strings and intervals. In *Proc. 21st ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010.

[16] A. Borsotti, L. Breveglieri, S. C. Reghizzi, and A. Morzenti. From ambiguous regular expressions to deterministic parsing automata. In *Implementation and Application of Automata*, pages 35–48. Springer, 2015.

[17] A. Borsotti, L. Breveglieri, S. C. Reghizzi, and A. Morzenti. BSP: A parsing tool for ambiguous regular expressions. In *Implementation and Application of Automata*, pages 313–316. Springer, 2015.

[18] B. Brodie, D. Taylor, and R. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. *ACM SIGARCH Computer Architecture News*, 34(2):202, 2006.

[19] A. Brüggemann-Klein and D. Wood. One-unambiguous regular languages. *Information and computation*, 140(2):229–253, 1998.

[20] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4): 481–494, 1964. ISSN 0004-5411.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill, 3d edition, 2009.

[22] L. D'Antoni and M. Veanes. Static Analysis of String Encoders and Decoders. In *VMCAI 2013*, volume 7737 of *LNCS*, pages 209–228. Springer Verlag, 2013.

[23] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, San Diego, California, January 2014. ACM Press.

[24] K. Ellul, B. Krawetz, J. Shallit, and M.-w. Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005.

[25] J. Engelfriet and H. Hoogeboom. MSO definable string transductions and two-way finite-state transducers. *ACM Transactions on Computational Logic (TOCL)*, 2(2):216–254, 2001. ISSN 1529-3785.

[26] K. Fisher and R. Gruber. PADS: a domain-specific language for processing ad hoc data. *ACM Sigplan Notices*, 40(6):295–304, 2005.

[27] K. Fisher and D. Walker. The PADS project: an overview. In *Proceedings of the 14th International Conference on Database Theory*, pages 11–17. ACM, 2011.

[28] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *ACM SIGPLAN Notices*, number 1 in 39, pages 111–122. ACM, 2004.

[29] A. Frisch and L. Cardelli. Greedy regular expression matching. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture notes in computer science*, pages 618–629, Turku, Finland, July 2004. Springer.

[30] J. Goyvaerts and S. Levithan. *Regular Expressions Cookbook*. O'Reilly, 2009. ISBN 978-0-596-52068-7.

[31] N. B. B. Grathwohl, F. Henglein, L. Nielsen, and U. T. Rasmussen. Two-pass greedy regular expression parsing. In *Proc. 18th International Conference on Implementation and Application of Automata (CIAA)*, volume 7982 of *Lecture Notes in Computer Science (LNCS)*, pages 60–71. Springer, July 2013.

[32] N. B. B. Grathwohl, F. Henglein, and U. T. Rasmussen. Optimally Streaming Greedy Regular Expression Parsing. In *Theoretical Aspects of Computing - ICTAC 2014 - 11th International Colloquium, Bucharest, Romania, September 17-19, 2014. Proceedings*, pages 224–240, 2014.

[33] C. Graulund. On automata-theoretic characterizations of regular expressions as types. Bachelor Thesis, Department of Mathematics, University of Copenhagen, May 2015.

[34] P. Hazel. PCRE – Perl-compatible regular expressions. Concatenation of PCRE man pages, January 3 2010.

[35] F. Henglein and L. Nielsen. Regular expression containment: Coinductive axiomatization and computational interpretation. *SIGPLAN Notices, Proc. 38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 46(1):385–398, January 2011. .

[36] T. Johnson, N. Robertson, P. D. Seymour, and R. Thomas. Directed tree-width. *Journal of Combinatorial Theory, Series B*, 82(1):138–154, 2001.

[37] S. Kearns. Extending regular expressions with context operators and parse extraction. *Software - Practice and Experience*, 21(8):787–804, 1991.

[38] K. Kosako. The Oniguruma regular expression library, 2014. URL `http://www.geocities.jp/kosako3/oniguruma/`.

[39] D. Kozen. *Automata and computability*. Springer Verlag, 1997.

[40] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *Journal of the ACM (JACM)*, 49(1):1–15, 2002.

[41] M. Lutz. *Programming Python*, volume 8. O'Reilly, 4th edition edition, December 2010. ISBN 978-0-596-15810-1.

[42] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Trans. on Electronic Comput.*, EC-9(1):38–47, 1960. . URL `http://dx.doi.org/10.1109/TEC.1960.5221603`.

[43] M. Mohri. Finite-state transducers in language and speech processing. *Computational linguistics*, 23(2):269–311, 1997.

[44] E. Myers, P. Oliva, and K. Guimarães. Reporting exact and approximate regular expression matches. In *Combinatorial Pattern Matching*, pages 91–103. Springer, 1998.

[45] E. W. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of mathematical biology*, 51(1):5–37, 1989.

[46] G. Myers. A four Russians algorithm for regular expression pattern matching. *J. ACM*, 39(2):432–448, 1992. ISSN 0004-5411.

[47] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 529–542. ACM, 2014.

[48] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. *Algorithm Engineering*, pages 1–13, 2001.

[49] L. Nielsen and F. Henglein. Bit-coded regular expression parsing. In *Proc. 5th Int'l Conf. on Language and Automata Theory and Applications (LATA)*, Lecture Notes in Computer Science (LNCS), pages 402–413. Springer, May 2011.

[50] S. Okui and T. Suzuki. Disambiguation in Regular Expression Matching via Position Automata with Augmented Transitions. In M. Domaratzki and K. Salomaa, editors, *Implementation and Application of Automata*, volume 6482 of *Lecture Notes in Computer Science*, pages 231–240. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-18097-2. . URL `http://dx.doi.org/10.1007/978-3-642-18098-9_25`.

[51] J.-F. Raskin and F. Servais. Visibly Pushdown Transducers. In L. Aceto, I. Damgård, L. A. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming*, volume 5126 of *Lecture Notes in Computer Science*, pages 386–397. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-70582-6. . URL `http://dx.doi.org/10.1007/978-3-540-70583-3_32`.

[52] A. Rathnayake and H. Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[53] M. Schützenberger. Sur une variante des fonctions sequentielles. *Theoretical Computer Science*, 4(1):47–57, Feb. 1977.

[54] R. Sidhu and V. Prasanna. Fast Regular Expression Matching Using FPGAs. In *Proc. 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2001. FCCM '01*, pages 227–238, 2001.

[55] G. Stewart, M. Gowda, G. Mainland, B. Radunovic, D. Vytiniotis, and C. L. Agulló. Ziria: A DSL for wireless systems programming. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 415–428. ACM, 2015.

[56] S. Sugiyama and Y. Minamide. Checking time linearity of regular expression matching based on backtracking. In *IPSJ Transactions on Programming*, number 3 in 7, pages 1–11, 2014.

[57] M. Sulzmann and K. Z. M. Lu. Regular Expression Sub-matching Using Partial Derivatives. In *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, pages 79–90, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1522-7.

[58] M. Sulzmann and K. Z. M. Lu. Posix regular expression parsing with derivatives. In *Proc. 12th International Symposium on Functional and Logic Programming*, FLOPS '14, Kanazawa, Japan, June 2014.

[59] J.-M. Talbot and P.-A. Reynier. Visibly Pushdown Transducers with Well-nested Outputs. Technical report, Aix Marseille Universite, CNRS, 2014. URL `https://hal.archives-ouvertes.fr/hal-00988129/`.

[60] The GNU Project, 2015. URL `http://www.gnu.org/software/coreutils/coreutils.html`.

[61] The Hyperscan authors. Hyperscan, October 2015. URL `https://01.org/hyperscan`.

[62] The RE2 authors. RE2, 2015. URL `https://github.com/google/re2`.

[63] The RE2J authors. RE2J, 2015. URL `https://github.com/google/re2j`.

[64] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968. ISSN 0001-0782. .

[65] A. Thurston. Ragel state machine compiler, 2015. URL `http://www.colm.net/open-source/ragel/`.

[66] G. van Noord and D. Gerdemann. Finite State Transducers with Predicates and Identities. *Grammars*, 4(3):263–286, 2001. ISSN 1386-7393.

[67] M. Veanes. Symbolic String Transformations with Regular Lookahead and Rollback. In *Ershov Informatics Conference (PSI'14)*. Springer Verlag, 2014.

[68] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual Symposium on Principles of Programming Languages*, POPL '12, pages 137–150, New York, NY, USA, 2012.

[69] M. Veanes, D. Molnar, T. Mytkowicz, and B. Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, 2015.

[70] P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, June 1990. ISSN 0304-3975.

[71] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3rd edition, July 2000.

[72] B. W. Watson. Implementing and using finite automata toolkits. *Natural Language Engineering*, 2(04):295–302, 1996. ISSN 1469-8110.

[73] B. B. Welch, K. Jones, and J. Hobbs. *Practical programming in Tcl and Tk*. Prentice Hall, 4th edition edition, 2003. ISBN 0130385603.

[74] S. Wu and U. Manber. Agrep–a fast approximate pattern-matching tool. *Usenix Winter 1992*, pages 153–162, 1992.

[75] L. Yang, P. Manadhata, W. Horne, P. Rao, and V. Ganapathy. Fast submatch extraction using OBDDs. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 163–174, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1685-9.