# Generalised Parikh Images of Regular Languages

some cool authors

Uppsala University, Sweden

**Abstract.** The image of the Parikh map is important in automata theory, offering a compact characterisation of an automaton. We contribute a novel understanding of how Parikh maps can be combined with arbitrary commutative monoid morphisms to efficiently represent a wide range of logics on automata and automata-like structures. Cases studied as examples include epistemic logic and string-length constraints in a string constraint solver. Furthermore, we show how this formulation can be efficiently implemented as a calculus in a theorem prover for succinct formulations of *several constraints on strings*. In particular, our calculus is versatile enough to efficiently compute the Parikh image of a product of two automata, allowing our solver, Ostrich, to solve *X new instances* where it was previously constrained by the memory required to materialise the product. Finally, we show that this implementation in addition to being *Z better* also offers *X performance* improvements on *Y real-world instances*, and in particular *yields no regressions* in performance *and, in fact, cures cancer, brings about world peace, and ends global hunger.*

## 1 Introduction

Parikh maps and their image appears naturally as part of many operations in model checking and the solving of string constraints in automata-based string solvers such as Ostrich. While it is possible to compute the Parikh image of an automaton using the method described in [2], this method produces large existentially quantified clauses which are costly to eliminate and in practice make many real-world problems intractable. Furthermore, many modern applications require automata with symbolic labels to handle large alphabets *citation needed*. They also compute Parikh images on products of automata, which would require the computation of the product before the Parikh image can be computed, running the risk of an exponential blow-up.

Addressing these concerns, our approach allows us to extend the computation of Parikh images to handle symbolic transition labels naturally, while also allowing us to interleave the computation of arbitrarily deep products of automata with the Parikh image of their product. This allows us to let both calculations inform each other, eliminating unnecessary work.

*It's also versatile as heck!*

### 1.1 Motivating Example

### 1.2 Related Work

## 2 Overview

Formally, the *Parikh map* over a context-free language $\Sigma = \{a_1, \ldots, a_k\}$ is defined as in [1]:

$$\psi : \Sigma^* \to \mathbb{N}^k$$
$$\psi(s) = [\#a_1(s), \#a_2(s), \ldots, \#a_k(s)]$$

That is, $\psi(s)$ is a vector of the number of occurrences of each character in the language for a given string $s$. For example, for $\Sigma = \{a, b\}$, we would have $\psi(abb) = [1, 2]$.

We define the image of this map, the *Parikh image*, of some subset of the language $A \subseteq \Sigma^*$ as:

$$\psi(A) = \{\psi(x) | x \in A\}$$

Thus we would have $\psi(\{ab, abb\}) = \{[1, 1], [1, 2]\}$.

**The Parikh image of a regular language expressed in Presburger arithmetic.** Following [2], we define the Parikh Image of a regular language recognised by a DFA $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$ as:

$$\psi(\mathcal{A}) := \bigwedge_{\alpha \in \Sigma} c_\alpha = \sum_{\delta \in \delta} t_\delta \text{ where } \alpha \in \delta$$
$$\bigwedge_{q \in Q} \left( 1 \text{ if } q \in I + \sum_{\delta = q' \to q} t_\delta - \sum_{\delta = q \to q'} t_\delta \right) \begin{cases} \geq 0 & \text{if } q \in I \\ = 0 & \text{otherwise} \end{cases}$$
$$\bigwedge_{\delta = q \to q'} t_\delta > 0 \implies z_{\prime q} > 0$$
$$\bigwedge_{q, q' \in Q} z_{q'} > 0 \implies \begin{cases} \bigvee\limits_{\delta = q \to q'} z_{q'} = z_q + 1 \wedge t_\delta > 0 \wedge z_q > 0 & \text{if } q \notin I \\ z_{q'} = 1 \wedge t_\delta > 0 & \text{if } q \in I \end{cases}$$

where all variables $z_i, t_i$ are existentially quantified and the variables $c_\alpha$ make up the actual image.

### 2.1 An Example

### 2.2 Generalised Parikh Images

Another way of viewing the Parikh map is as a monoid homomorphism $p : (\Sigma^*, \cdot, \epsilon) \to (\mathbb{Z}^\Sigma, +, \vec{0})$, where $\cdot$ is the string concatenation operation, the objects

of the right-hand-side monoid are character counts, and + is standard vector addition. Note that while the left monoid does not commute, the right one does.

This viewpoint enables us to generalise the Parikh map and its image further to arbitrary monoid morphisms $h : \Sigma^* \to M$ where $M$ is a commutative monoid. It then follows from the universal mapping property that any such morphism $h$ can also be expressed in terms of the Parikh map, as $h' \circ p$.

A useful example of such a morphism might be computing the length of a string, which could be recast in terms of the Parikh map by summing the individual character counts of the vector: $h' : (\mathbb{Z}^\Sigma, +, \vec{0}) \to (\mathbb{Z}, +, 0) = \vec{x} \to \sum_{i \in \Sigma} x_i$, where the respective operators + is element-wise vector addition and integer addition respectively.

**An Example**

### 2.3 Lazy Computation of Parikh Images for Regular Languages

- Lazy expansion of the Parikh conditions for a symbolic automaton
- Finding elements vs. computing the complete Parikh image
- Lazy product computation

- Preliminaries, the underlying calculus, what are rules
- Predicates used to represent Parikh images
- Our calculus rules
- Statement of properties, correctness, complexity

- remove the monoid map!!!

We assume an NFA $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$ with $t$ transitions $\delta = \{\delta_1, \ldots, \delta_t\}$ where we describe each such transition $\delta_i$ from node $q$ to node $q'$ with label $l \in \Sigma$ as $\delta_i = q \xrightarrow{l} q'$.

Treating $\mathcal{A}$ as a graph with vertices $Q$ and edges $\delta$, we use the term *separating cut* of a set of *transitions* $T = \{\delta_i, \ldots, \delta_n\}, \mathrm{Cut}(T)$ to refer to any set of transitions whose removal causes $T$ to be unreachable from any state in $I$, with the meaning that $e = v \xrightarrow{l} v'$ is reachable if $v$ is. Note that if $T$ contains a transition such that $v \in I$, $\mathrm{Cut}(T) = \emptyset$, since we cannot disconnect an initial state by removing any transition.

We will follow the notation of [2] and simultaneously talk about $\mathcal{A}$ as a graph and an automaton. Moreover, we will continuously refer to the subgraph produced by keeping only the transitions/edges whose corresponding variables in $x$ are positive ($> 0$). An edge will be called *selected* if it is in this subgraph. An edge that is known to be zero will conversely be called *deselected*. An edge whose corresponding term has no known status is called unknown. Formally, we define these as follows:

TODO what happens to $x$ now that we have TM?

TODO how do we work around the presburger logic here?

**Definition 1.** $\textsc{Selected}(\mathcal{A}, t)(\phi)$ *means that* $\phi$ *contains* $\mathrm{TM}(\mathcal{A}, t, x)$ *and that* $\phi \vDash_{\textsc{Presburger}} x > 0$.

**Definition 2.** $\textsc{Deselected}(\mathcal{A}, t)(\phi)$ *means that* $\phi$ *contains* $\mathrm{TM}(\mathcal{A}, t, x)$ *and that* $\phi \vDash_{\textsc{Presburger}} x \leq 0$.

**Definition 3.** $\textsc{Unknown}(\mathcal{A}, t)(\phi) = \neg\,\textsc{Selected}(\mathcal{A}, t)(\phi) \wedge \neg\,\textsc{Deselected}(\mathcal{A}, t)(\phi)$, *while* $\mathrm{TM}(\mathcal{A}, t, x)$ *is still a term in* $\phi$.

For all of these cases, we say that a transition $t$ is $\textsc{Selected}, \textsc{Deselected}$ or $\textsc{Unknown}$ respectively, usually leaving out the clause $\phi$ and automaton $\mathcal{A}$. Additionally, we define two helper predicates, one of which we have already used:

**Definition 4.** $\textsc{Conn}(\mathcal{A})$ *holds when every* $\textsc{Selected}$ *or* $\textsc{Unknown}$ *transition in* $\mathcal{A}$ *is reachable from its initial state.*

**Definition 5.** $\mathrm{TT}(\mathcal{A}, t, x)$ *holds when* $\mathcal{A}$*'s transition* $t$ *is taken* $x$ *times (an integer value).*

With these preliminaries out of the way, we can define our main predicate:

**Definition 6.** $\mathrm{Im}_{\mathcal{A},h}(x, y)$ *is true exactly when:*

- $h : \Sigma^* \to M$*, a morphism to a commutative product monoid* $M = \prod_i M_i$.
- $y$ *is an element of* $M$.
- $x$ *is a vector of terms such that each term correspond to a transition in* $\mathcal{A}$.

To simplify equations, we let $h(t)$ for some transition $t \in \delta$ to mean the application of $h$ to $t$'s label. Similarly, we use the notation $x_t$ to refer to transition $t \in \delta$'s corresponding term, allowing $x$ to be directly indexed by the transitions w.l.o.g from integer indices. We will also consistently assume that we have only one instance of our main predicate. In the actual implementation, described in *the north pole*, we will use an additional, existentially quantified, integer variable to identify the instance of a predicate.

$$\textsc{Propagate}\ \frac{\bigwedge_{\delta_t \in T} x_{\delta_t} = 0 \wedge \mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \bigwedge_{\delta_c \in C} x_{\delta_c} = 0 \wedge \phi}{\mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \bigwedge_{\delta_c \in C} x_{\delta_c} = 0 \wedge \phi}\ C = \mathrm{Cut}(T),\ \exists t \in T\ :\ \neg\,\textsc{Deselected}(t)$$

$$\textsc{Expand}\ \frac{\textsc{Flow-Eqs}(\mathcal{A}) \wedge \textsc{Conn}(\mathcal{A}) \wedge y = \sum_{t \in \delta} x_t \cdot_M h(t) \wedge \bigwedge_{i \in 1,\ldots,t} x_i \geq 0 \wedge \mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \phi}{\mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \phi}\ \text{Precisely once}$$

$$\textsc{Split}\ \frac{x_i = 0 \wedge \mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \phi \quad | \quad x_i > 0 \wedge \mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \phi}{\mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \phi}\ \text{if } \textsc{Unknown}(x_i)(\phi)$$

$$\textsc{Subsume-Main}\ \frac{\phi}{\mathrm{Im}_{\mathcal{A},h}(x,y) \wedge \phi}\ \text{if no instances of } \textsc{Conn}(\mathcal{A}) \text{ in } \phi$$

$$\textsc{Subsume-Connected}\ \frac{\phi}{\textsc{Conn}(\mathcal{A}) \wedge \phi}\ \text{if } \textsc{Known-Connected}$$

where $\textsc{Flow-Eqs}(\mathcal{A})$ are the flow-balancing part of the Parikh image from Section 2.2 for an automaton $\mathcal{A} = \langle Q, \Sigma, \delta, I, F \rangle$:

$$\textsc{In}(q) = 1 \text{ if } q \in I + \sum_{i \in 1,\ldots,t \mid \delta_i = * \rightarrow q} x_i$$

$$\textsc{Out}(q) = \sum_{i \in 1,\ldots,t \mid \delta_i = q \rightarrow *} x_i$$

$$\textsc{Flow-Eqs} = \bigwedge_{q \in Q} \textsc{In}(q) - \textsc{Out}(q) \geq 0 \text{ if } q \in F, = 0 \text{ otherwise}$$

and Known-Connected corresponds to the following, implying guaranteed connectedness (or, conversely, the non-existence of cuts disagreeing with $x$):

$$\forall C, T\ C = \mathrm{Cut}(T) \implies \forall i\ \delta_i \in T \wedge x_i > 0 \implies \forall j\ \delta_j \in C \implies x_j = 0$$

Additionally, we assume the existence of a rule Presburger-Close, corresponding to a sound and complete solver for Presburger formulae.

The Propagate rule allows us to propagate connectedness across $\mathcal{A}$. It states that we are only allowed to "use" transitions attached to a reachable state, and is necessary to ensure connectedness in the presence of cycles. Expand expands the predicate into its most basic rules; one set of linear equations connecting $x$ and $y$, and the linear flow equations of the standard Parikh image formulation.

Finally, Split allows us to branch on the proof tree by first trying to exclude a contested edge from a potential solution and then concluding that it must be included.

A decision procedure for our predicate in a tableau-based automated theorem prover would start by expanding the predicate using the Expand rule. For many instances of the predicate, this would be enough to induce subsumption; as long as the DFA contains no loops that could be disconnected from a minimum spanning tree (MST) of the automaton.

### 2.4 An Example

## 3 Generalising the calculus modulo arbitrary commutative monoid maps

### 3.1 An Example

## 4 Parikh Images from Products of Automata

- additional rules needed for products
- backjumping and conflict-driven learning

To generalise the calculus to calculations on products of automata, we change the main predicate take arbitrarily many automata:

**Definition 7.** $\Pi(\mathbf{A}, \mathbf{Tr}, y)$ *is true exactly when:*

- $h : \Sigma^* \to M$*, a morphism to a commutative product monoid* $M = \prod_i M_i$*.*
- $y$ *is a vector of elements of* $M$*.*
- $\mathbf{A}$ *is a vector of automata.*
- $\mathcal{A} = \prod_i \mathbf{A}_i$
- *each* $x_i$ *is a vector of terms such that each term correspond to a transition in* $\mathbf{A}_i$*, and* $\mathbf{Tr}$ *is a vector of those vectors. We use the terminology* $\mathrm{TT}(t)$ *to refer to the term of the transition vector for a given transition* $t$*.*
- $y = \sum_{t \in \delta} h(t)$
- $\mathrm{Source}(\mathcal{A}, t)$ *is the set of transitions from the term automata used to produce the transition* $t$ *in the product automaton* $\mathcal{A}$*.*

First we extend Expand to generate flow equations and instances of Conn for each automaton:

$$\textsc{Expand}\ \frac{\bigwedge_{\mathcal{A}_i \in \mathbf{A}} \textsc{Flow-Eqs}(\mathcal{A}_i) \wedge \textsc{Conn}(\mathcal{A}_i) \ldots}{\Pi(\mathbf{A}, \mathbf{Tr}, y) \wedge \phi}$$

Then we introduce the rule Materialise, used to compute a partial product between two terms $\mathcal{A}_i, \mathcal{A}_j$:

$$\textsc{Materialise}\ \frac{\bigwedge_{t \in \delta(\mathcal{A}_k)} \mathrm{TT}(t) = \sum_{t' \in \mathrm{Source}(t, \mathcal{A}_k)} t' \quad \textsc{Flow-Eqs}(\mathcal{A}_k) \wedge \textsc{Conn}(\mathcal{A}_k) \wedge \Pi(\mathbf{A}', \mathbf{Tr}', y) \wedge \phi}{\Pi(\mathbf{A}, \mathbf{Tr}, y) \wedge \phi} \text{ where } \mathcal{A}_k = \mathcal{A}_i \times \mathcal{A}_j, \mathbf{A}' = (\mathbf{A} \cup \{\mathcal{A}_k\}) \setminus \mathcal{A}_i, \mathcal{A}_j$$

Note that we implicitly existentially quantify the fresh transition terms $x_{k+1}$ for the product.

Materialise comes with its companion rule NoConn that simply states we are allowed to remove any instances of Conn from terms that have been used in a product, and whose connectedness we therefore no longer care about:

$$\textsc{NoConn}\ \frac{\Pi(\mathbf{A}, \mathbf{Tr}, y) \wedge \phi}{\textsc{Conn}(\mathcal{A}_a) \wedge \Pi(\mathbf{A}, \mathbf{Tr}, y) \wedge \phi} \text{ If } \mathcal{A}_a \text{ not in } \mathcal{A}_1 \ldots \mathcal{A}_k$$

Finally, we add a side condition to Subsume to only allow subsumption for instances of $\Pi$ with just one automaton, ensuring that we do not bail out before having computed the entire product.

### 4.1 An Example

## 5 Solving the Travelling Salesperson Problem

If we take an instance of the (decision version of the) TSP as a graph with vertices $N$, edges $E$ and a distance function $\delta(e)$, we can solve it using our method in the following way:

1. Construct an automaton $\mathcal{A}$ such that all its states are accepting, all the vertices and edges from the TSP graph are present, and the initial state has a transition to each state with weight 0. This encodes the movements in the graph, capturing the constraint that we may take any path through the graph, and that we may end anywhere.
2. Use a monoid morphism to an $n+1$-element vector, where the first element is the length of the current path, and the $n = |N|$ elements following it is how many times we visit each node.
3. The monoid morphism $m(\langle a, w, b \rangle) = [w, 0, \ldots, 1, \ldots, 0]$ would simply map to the transition weight $w$ followed by a 1 for node $b$ and zeroes for all other nodes. The corresponding (commutative) monoid operation would then be simple elementwise addition of the bookkeeping vector.
4. Constrain the terms corresponding to the visited status for each state to be at least 1.
5. Iteratively query the solver for a looser and looser upper bound on the length. Once we have a solution, we know it to be minimal.

## 6 Implementation and Experiments

– length constraints
– Parikh automata, automata with registers
– Model-checking examples

## 7 Conclusions

## References

[1] Dexter C. Kozen. *Automata and computability*. New York: Springer, 1997. ISBN: 0387949070.

[2] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. "On the Complexity of Equational Horn Clauses". In: *Automated Deduction – CADE-20*. Ed. by Robert Nieuwenhuis. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 337–352. ISBN: 978-3-540-31864-4.