Semi-linear Parikh Images of Regular Expressions via Reduction

Bahareh Badban[1] and Mohammad Torabi Dashti[2]

[1] Universität Konstanz
[2] ETH Zürich

Abstract. A reduction system for regular expressions is presented. For a regular expression t, the reduction system is proved to terminate in a state where the most-reduced expression readily yields a semi-linear representation for the Parikh image of the language of t.

1 Introduction

Let $A = \{a_1, \cdots, a_n\}$ be a finite set, and fix an order on the elements of A, say $a_1 < \cdots < a_n$. The *commutative* image, or *Parikh* image, of $w \in A^*$, denoted $\Psi(w)$, is a vector in $\mathbb{N}^n$ in which the i^{th} element encodes how many times a_i appears in w. Hence, Ψ is a morphism from the monoid $(A^*, \cdot, \epsilon)$ to the monoid $(\mathbb{N}^n, +, \mathbf{0})$. The commutative image of any regular subset of A^* (in general, any context-free language) is a *semi-linear* set [6], i.e. it can be written as the union of finitely many *linear* sets. A linear set is of the form $\{v \mid v = v_0 + \lambda_1 v_1 + \cdots + \lambda_k v_k, \text{ with } k, \lambda_i \in \mathbb{N}, v_i \in \mathbb{N}^n\}$. In this paper, we study the following problem:

> Given a regular expression t, compute a semi-linear representation for the Parikh image of the regular language that t denotes.

Contributions. A terminating reduction system is proposed which reduces a regular expression t to a simpler regular expression t', with certain constraints, such that t and t' have the same Parikh image. The reduction system is proved to terminate in a state where the resulting most-reduced expression is of star-height at most one. A semi-linear representation of the Parikh image of the initial expression t is then computed from the most-reduced expression by resolving the accumulated constraints.

As an example, consider the regular expression $t = (a^* \cdot b)^*$, with alphabet $A = \{a, b\}$, and $a < b$. In order to compute a semi-linear representation for $\Psi(t)$, we introduce the notion of *constrained regular expressions*. In the example above, we reduce t to $(a^*)^* \cdot b^*$, which is in turn reduced to $a^* \cdot b^*$, while a constraint is added to the expression asserting that "if there is no b, then there is no a". Intuitively, adding constraints to regular expressions allows us to encode the dependency between nested Kleene stars. The reduction system then propagates these constraints downwards while descending and pushing the Kleene stars

P. Hliněný and A. Kučera (Eds.): MFCS 2010, LNCS 6281, pp. 653–664, 2010.

down the parse tree of the regular expression. This procedure eventually leads to a constrained regular expression of star-height at most one. In a second phase, the accumulated constraints are resolved into linear conditions using case distinction. In the example above, either there is no b, and therefore no a, or there is at least one b and an arbitrary number of as. The resulting constraints are linear, hence their union semi-linear.

Related work. Algorithms for computing semi-linear Parikh images of regular grammars (hence, regular expressions) can be extracted from Parikh's original proof [6]. However, Parikh's proof, as well as some of the later variants [4,2], do not specifically aim at *constructing* semi-linear images. For instance, in [6], after a careful rearrangement of derivation trees for context-free grammars one reads: "We claim there is only a finite number of trees satisfying [a certain condition], since in any such tree the length of any chain cannot be greater than the square of the number of symbols [of the grammar]". Intuitively, this proof relies on finiteness of a certain set, and in effect, derives an upper bound on the number of possible elements in another set.

Roughly speaking, semi-linear Parikh images of regular expressions can be constructed via two different strategies: *bottom-up* and *top-down*.

- The bottom-up strategy is perhaps the first method that comes to mind. Start with the symbols of the alphabet, and proceed inductively: Assume that semi-linear images of t_1 and t_2 are given, and use the following rules for computing semi-linear images of more complex expressions.
  - $\Psi(t_1 \cdot t_2) = \Psi(t_1) \oplus \Psi(t_2)$, where $A \oplus B = \{a + b \mid a \in A, b \in B\}$ for two sets A and B.
  - $\Psi(t_1 \cup t_2) = \Psi(t_1) \cup \Psi(t_2)$
  - $\Psi(t_1^*) = S_1^* \oplus \cdots \oplus S_r^*$, if $\Psi(t_1) = S_1 \cup \cdots \cup S_r$, with S_i being linear sets, and $S_i^* = \{\mathbf{0}\} \cup \{v \mid v = v_0 + \lambda_0 v_0 + \lambda_1 v_1 + \cdots + \lambda_k v_k, \text{ with } k, \lambda_i \in \mathbb{N}, v_i \in \mathbb{N}^n\}$ if $S_i = \{v \mid v = v_0 + \lambda_1 v_1 + \cdots + \lambda_k v_k, \text{ with } k, \lambda_i \in \mathbb{N}, v_i \in \mathbb{N}^n\}$.

  The inductive method gives a terminating algorithm for computing semi-linear Parikh images, e.g. see [3] for a correctness proof for these rules.
- The main idea of the top-down strategy is to obtain from regular expression t a regular expression t' which has star-height of at most one, while $L(t) = L(t')$ modulo commutativity. Deriving semi-linear Parikh images of the languages of regular expressions of star-height at most one is immediate. A prominent example of the top-down approach is the equational axioms of Piling [7], which can be used to convert any regular expression t to an equivalent regular expression t', modulo commutativity, such that t' is of star-height at most one. See [1] for an excellent overview. These equational axioms however do not immediately lend themselves to a terminating algorithm for computing semi-linear Parikh images. This is because the axioms contain the equation $x \cdot y = y \cdot x$, among others. This equation can cause a cyclic behavior, and hence non-termination, in automated procedures that use it for reducing the star-height of regular expressions.

Our reduction system also follows the top-down strategy. What distinguishes the proposed reduction system from the aforementioned equational axioms is termination – the reduction system is proved to always terminate. The difference between the reduction system and the (bottom-up) inductive method however lies not in termination, but in the *width* of the produced images.

Consider a semi-linear set S represented as $S = S_1 \cup \cdots \cup S_\ell$, with $S_{i \in 1..\ell}$ being linear sets. The width of S, denoted $\omega(S)$, is the number of linear components in S, that is ℓ in this case. Obviously a semi-linear set may have different representations with different widths. The width of semi-linear representations for Parikh images of regular expressions is pertinent to certain decision procedures for regular languages, e.g. see [5,10]. In these procedures, various properties of regular languages are decided by solving a linear Diophantine equation for each linear component of the semi-linear Parikh image of the language. In this context, images with lower width are preferred, because then fewer Diophantine equations need to be solved for answering the decision problem at hand.

The width of the semi-linear representation that the aforementioned inductive method produces for regular expression t grows fast as the widths of the images of the subterms of t grow. This is intuitively due to the "multiplicative" nature of the inductive method. Suppose semi-linear representations for Parikh images of regular expressions t_1 and t_2 are given, resp. with widths w_1 and w_2. The width of the representations that the inductive method produces can then be calculated as $\omega(\Psi(t_1 \cdot t_2)) = w_1 w_2$, $\omega(\Psi(t_1 \cup t_2)) = w_1 + w_2$ and $\omega(\Psi(t_1^*)) = 2^{w_1}$.

Our proposed reduction system produces semi-linear Parikh images which are of (exponentially) lower width, compared to the inductive method. As a simple example, for the Parikh image of the regular expression $a^* \cdot (b \cup c^*)^* \cdot d$, with alphabet $A = \{a, b, c, d\}$, the reduction system produces a semi-linear representation of width 2, while the inductive method produces an image of width 16. Formally, for *unit form* regular expressions t_1 and t_2, whose images have widths w_1 and w_2 respectively, the width of the representations that the reduction system produces is given by $\omega(\Psi(t_1 \cdot t_2)) = w_1 w_2$, $\omega(\Psi(t_1 \cup t_2)) = w_1 + w_2$ and $\omega(\Psi(t_1^*)) = 2w_1$. The definition of unit form expressions, as well as precise bounds for the widths of Parikh images that the reduction system produces are given in the following sections.

Note that, w.r.t. the width of produced Parikh images, the equational axioms of [7,1] cannot be directly compared to the inductive method and the reduction system. This is because, in [7,1], depending on the axioms which are chosen for simplification, and their order, the resulting images may have different widths.

In a related work, Verma, Seidl and Schwentick give a linear-time algorithm for generating semi-linear Parikh images of context-free grammars, represented as existential Presburger formulae [9]. Converting existential Presburger formulae into the set representation that we use is of exponential-time complexity.

Structure of the paper. Section 2 presents the preliminaries. Section 3 describes our reduction system, and contains the proofs of its termination and correctness. Extracting semi-linear representations of Parikh images from the most-reduced expressions is explained in section 4.

2 Preliminaries

An *alphabet* is a finite set $A = \{a_1, \cdots, a_n\}$. Throughout the paper we assume alphabets are totally (and, lexicographically) ordered. The set of all finite sequences of elements of A is denoted by A^*, while ϵ stands for the empty sequence. A *language* over A is a subset of A^*. For two sequences $w_1, w_2 \in A^*$, let $w_1 \cdot w_2$ denote the concatenation of w_1 and w_2. Clearly ϵ is the neutral element of concatenation. For two sets $L_1, L_2 \subseteq A^*$, define $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2\}$. For $L \subseteq A^*$, we define $L^0 = \{\epsilon\}$ and $L^n = L \cdot L^{n-1}$, with $n \in \mathbb{N}, n > 0$.

A *regular expression* t, for short *regexp*, over A is defined as usual, $t ::= \emptyset \mid \epsilon \mid a \mid t \cdot t \mid t \cup t \mid t^*$, with $a \in A$. Here $\cdot$, $\cup$ and * denote, respectively, concatenation, union, and Kleene star operators. Any regular expression t defines a *regular language*, $L(t)$, in the standard way: $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$, $L(a) = \{a\}$ for all $a \in A$, $L(t_1 \cdot t_2) = L(t_1) \cdot L(t_2)$, $L(t_1 \cup t_2) = L(t_1) \cup L(t_2)$, and $L(t^*) = \bigcup_{i \geq 0} L^i$. We write $t \approx t'$ for regexps t and t' iff $L(t_1) = L(t_2)$. When confusion is unlikely we may use a regexp t and its corresponding language $L(t)$ interchangeably.

Any regular expression has a unique parse tree, up to isomorphism. Nodes of a parse tree are either elements of the alphabet (besides ϵ and $\emptyset$), or the symbols $\cdot$, $\cup$ or *. Nodes labeled with elements of the alphabet (and ϵ and $\emptyset$) are terminal, nodes labeled with either of $\cdot$ or $\cup$ have two descendants, while nodes labeled with * have only one descendant. The parse trees of regular expressions correspond to the ground term algebra induced by the signature *Sig*: the elements of the alphabet are nullary function symbols in *Sig*, $\cdot$ and $\cup$ are binary function symbols, and * is a unary function symbol.

A regexp is said to be of *unit form*, for short $\mathcal{U}$-form, iff in its parse tree each path from the root to a node labeled with $\cup$ contains at least one node labeled with *. The following lemma is immediate as $\cdot$ is distributive over $\cup$ modulo $\approx$.

Lemma 1. *Any regexp t can be rewritten into finitely many regexps $t_1, \cdots, t_n$, such that $t \approx t_1 \cup t_2 \cup \cdots \cup t_n$, with t_i being of $\mathcal{U}$-form.*

For $L \subseteq A^*$, let $\Psi(L) = \{\Psi(w) \mid w \in L\}$. When computing semi-linear Parikh images of languages of regexps, lemma 1 allows us to confine to $\mathcal{U}$-form expressions. This is because $\Psi(L_1 \cup L_2) = \Psi(L_1) \cup \Psi(L_2)$. In the following, thus, we focus on $\mathcal{U}$-form regexps.

Below, we define a simple extension of regexps which is useful in proving our results. Fix a countable set of *name tags* $\mathsf{Names} = \{n_1, n_2, \ldots\}$. A *tagged* regexp t is a regexp where some of the nodes labeled with * in the parse tree of t are assigned with elements of Names. Note that any regexp is a tagged regexp, where *none* of its parse tree nodes are tagged. As a convention, in the text we subscript a * operator of a tagged regexp with its name. For instance, suppose the node labeled with * in the parse tree of $(a \cdot (b \cup c))^*$ is assigned with n_1. We denote this by $(a \cdot (b \cup c))^{*_{n_1}}$. An *evaluation* function is a total function $e : \mathsf{Names} \to \mathbb{N}$. For a tagged regexp t, we write t_e for the regexp that corresponds to t when all *s assigned with n are replaced with $e(n)$, for all $n \in \mathsf{Names}$. For example, with $e(n_1) = 2$ and $t = (a \cdot (b \cup c))^{*_{n_1}}$, t_e corresponds to regexp $(a \cdot (b \cup c))^2$. Recall that $r^k = r \cdot r^{k-1}$ for $k \geq 1$, and $r^0 = \epsilon$, for regexp r.

A *constrained* regexp is a tuple $(t;\phi)$, where t is a tagged regexp, and ϕ is a finite set of *constraints*. Each constraint is of either of these forms: $n_i + n_j = n_k$, or $n_i = 0 \implies n_j = 0$, with $n_i, n_j, n_k \in \mathsf{Names}$.

Given an evaluation function $e : \mathsf{Names} \to \mathbb{N}$, and a set of constraints $\phi(n_1, \cdots, n_\ell)$, we say e satisfies ϕ, denoted $e \models \phi$, iff all the constraints in ϕ evaluate to true when each $n \in \mathsf{Names}$ in ϕ is substituted with $e(n)$. That is, $\phi(e(n_1), \cdots, e(n_\ell))$ evaluates to true. The language of a constrained regexp $(t;\phi)$, denoted $L(t;\phi)$, is the set of all $w \in A^*$ where there exists an evaluation function e such that $w \in t_e$ and $e \models \phi$. We observe that $(t;\phi)$ can in general correspond to a non-regular language.

The example below shows that simply assigning names to * nodes of a regexp can affect the language of the corresponding constrained regexp, even if those names are not bound by any constraint.

Example 1. Consider the constrained regexps $(t;\emptyset)$, with $t = (a \cdot b^{*_{n_1}})^{*_{n_2}}$. Here the constraint set is empty. Note that $w = a \cdot b \cdot a \cdot b \cdot b$ does *not* belong to the language of $(t;\emptyset)$, while obviously w does belong to the language of $(a \cdot b^*)^*$. •

3 The Reduction System

Fix a signature S, and a countable set of variables $\mathcal{V}$. The term algebra induced by S with variables $\mathcal{V}$ is denoted by $\mathcal{T}_{S(\mathcal{V})}$. We write $var(t)$ for the set of variables appearing in term t. A *ground* term is an element of $\mathcal{T}_{S(\emptyset)}$.

A reduction system R is a finite set of ordered pairs, written as $l \to r$, where $l, r \in \mathcal{T}_{S(\mathcal{V})}$, and $var(r) \subseteq var(l)$. Let $l \to r \in R$ and $t \in \mathcal{T}_{S(\emptyset)}$ be a ground term. If for some substitution σ, $\sigma(l)$ is a subterm of t, then we can *reduce* t by replacing one occurrence of $\sigma(l)$ with $\sigma(r)$ in t. If the resulting term is t', we write $t \to t'$ and call it a *reduction step*. A term t is *irreducible* if $\neg\exists t'.\ t \to t'$; otherwise t is *reducible*. A *reduction sequence* is a sequence $t_1, t_2, \ldots$, such that $t_i \to t_{i+1}$, for $i \geq 1$. A reduction system is *terminating* iff it admits no infinite reduction sequences. See [8] for more on reduction systems.

Below, we introduce $\mathcal{R}$, our set of reduction rules for constrained regexps, that is the signature of $\mathcal{R}$ is *Sig* introduced in the previous section. As already mentioned, the ground term algebra $\mathcal{T}_{Sig(\emptyset)}$ corresponds to the set of all parse trees of regular expressions. Therefore, the following reduction rules are described in terms of parse trees. This greatly improves the presentation. The definitions and proofs can however be formalized in the corresponding term algebra as well.

The following reduction rules primarily work on parse trees of regexps of $\mathcal{U}$-form. If a reduction rule is *applicable* on an expression t, i.e. $t \to t'$, then the rule transforms the constrained regexp $(t;\phi)$ into $(t';\phi')$. In the following, how ϕ' is constructed using ϕ is also specified.

Definition 1 (Reduction rules $\mathcal{R}$). *The reduction rules depicted in figure 1 will be applied on constrained regexps $(t;\phi)$ where t is of $\mathcal{U}$-form. Here, in rules R_2 and R_3 fresh name tags are retrieved from* Names *as n_1 and n_2, and n' respectively.*

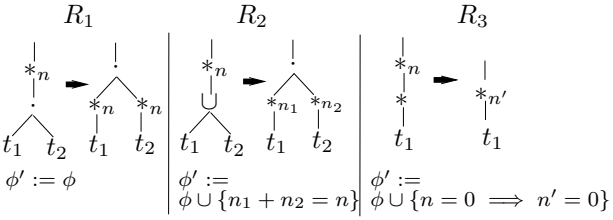

Fig. 1. Reduction rules; name tags of $*$ operators are written beside them

Intuitively, the rules of the reduction system $\mathcal{R}$ eliminate * functions that are applied on other functions (that is $\cdot, \cup$, and other *). Informally, rule (1), that is R_1, states $(t_1 \cdot t_2)^*$ can be rewritten to $t_1^* \cdot t_2^*$. The newly created stars (on top of t_1 and t_2) are assigned with the same name as the initial star. No constraint is added to ϕ in this case. Rule (2) states that $(t_1 \cup t_2)^*$ can be rewritten to $t_1^* \cdot t_2^*$. The newly created stars (on top of t_1 and t_2) are assigned with new names. A constraint is added to the constraint set ϕ, which requires the sum of the new names to be equal to the name of the initial star. Finally, rule (3) states that from $(t_1^*)^*$ the higher star can be removed. A constraint is however added to ϕ which requires if the name of the higher star equals zero, the name of the lower one should also be equal to zero.

Remark that the reduction rules of $\mathcal{R}$ can be consequently applied, because if t is of $\mathcal{U}$-form, and $t \to t'$ according to $\mathcal{R}$, then t' is of $\mathcal{U}$-form.

Applying $\mathcal{R}$ on $\mathcal{U}$-form regular expressions. Given a regexp t, we first *seed* t; this results in a constrained regexp. Then $\mathcal{R}$ is applied on the resulting constrained regexp, following the *top-most* reduction strategy. These notions are described below.

Seeding A node in the parse tree of t is called a *-node iff the node is labeled with symbol *. Given a $\mathcal{U}$-form regexp t, all the nodes in the parse tree of t which are labelled with * and have no ancestor *-node are assigned with fresh name tags. By abuse of notation, we write t also for the resulting tagged regexp. Then, the constrained regexp $(t; \emptyset)$ is constructed. This process is called seeding t. Note that the language of $(t, \emptyset)$ right after seeding is the same as the language of regexp t.

Top-most reduction strategy We assume a top-most reduction strategy, i.e. in the parse tree of a regexp the top-most reducible subterm of t is reduced first. The top-most reduction strategy corresponds to the *left-most* reduction strategy in the term algebra $\mathcal{T}_{Sig(\emptyset)}$, cf. [8]. The correctness of the reduction system, proved in theorem 3, assumes this reduction strategy.

We proceed with an example.

Example 2. Fix the alphabet $A = \{a, b, c\}$. Below we demonstrate how $\mathcal{R}$ is applied on $t = ((b \cdot a^*) \cup c)^*$. Note that t is indeed of $\mathcal{U}$-form. Seeding t yields

the pair $((b \cdot a^*) \cup c)^{*_n}; \emptyset$. Applying the top-most reduction strategy results in the following reduction sequence.

$$\begin{array}{lll} & ((b \cdot a^*) \cup c)^{*_n} & ; \emptyset \\ \rightarrow_{R_2} & (b \cdot a^*)^{*_{n_1}} \cdot c^{*_{n_2}} & ; \{n = n_1 + n_2\} \\ \rightarrow_{R_1} & b^{*_{n_1}} \cdot (a^*)^{*_{n_1}} \cdot c^{*_{n_2}} & ; \{n = n_1 + n_2\} \\ \rightarrow_{R_3} & b^{*_{n_1}} \cdot a^{*_{n_3}} \cdot c^{*_{n_2}} & ; \{n = n_1 + n_2,\ n_1 = 0 \implies n_3 = 0\} \end{array}$$ •

3.1 Termination

The reduction system $\mathcal{R}$ is terminating when applied to seeded $\mathcal{U}$-form regexps, intuitively because by applying each of the rules of $\mathcal{R}$, the * operators are pushed down the parse tree, see figure 1. Ultimately there will be * operators whose operands are constants (elements of A, or the symbols ϵ and $\emptyset$). These cannot be further reduced according to $\mathcal{R}$; the reduction process thus halts.

Theorem 1. *The reduction system $\mathcal{R}$ is terminating for $\mathcal{U}$-form regexps.*

For proving this theorem we need a few definition and lemmas. First, we define a mapping $\aleph$ from the set of regular expressions, over alphabet A, to $\mathbb{N}^3$. Intuitively, $\aleph$ counts the number of non-reduced subterms of a regexp, w.r.t. $\mathcal{R}$.

Definition 2. *Function $\aleph$ from the set of regexps over alphabet A to $\mathbb{N}^3$ is defined recursively by:*

$$\begin{array}{ll} \aleph(\emptyset) = \aleph(\epsilon) = \aleph(a) = \aleph(\epsilon^*) = \aleph(a^*) & = (0,0,0) \\ \aleph(t \cup t') = \aleph(t \cdot t') & = \aleph(t) + \aleph(t') \\ \aleph((t \cdot t')^*) & = (1,0,0) + \aleph(t^*) + \aleph(t'^*) \\ \aleph((t \cup t')^*) & = (0,1,0) + \aleph(t^*) + \aleph(t'^*) \\ \aleph((t^*)^*) & = (0,0,1) + \aleph(t^*) \end{array}$$

where $a \in A$. For any expression t, norm *of t is defined as $||t|| = \sum_{i=1}^{3} \aleph_i(t)$, where $\aleph_i(t)$ is the i^{th} element of the $\aleph(t)$-triplet.*

The reduction system $\mathcal{R}$ strictly reduces the norms of regexps. This is proved in the following lemma. Remark that $\aleph$ is agnostic to the tagging applied to regexps. Indeed, tagging plays no role in proving termination of $\mathcal{R}$.

Lemma 2. *If $t \rightarrow t'$ by the reduction system $\mathcal{R}$, then $||t'|| < ||t||$.*

Proof. The proof is by case analysis. Imagine $t = (r^*)^*$ for some regexp r. Observe that $\aleph_3(t) \geq 1$, and clearly R_3 of $\mathcal{R}$ is applicable on t. Using R_3, t is reduced to r^*, that is $t' = r^*$. Here $||t|| > ||t'||$, because $||t|| = \Sigma_{i=1}^{3} \aleph_i((r^*)^*) = 1 + \aleph_3(r^*) + \Sigma_{i=1}^{2} \aleph_i(r^*) = 1 + \Sigma_{i=1}^{3} \aleph_i(r^*) = 1 + ||r^*|| = 1 + ||t'||$. Similarly, $||(t_1 \cdot t_2)^*|| = 1 + ||t_1^* \cdot t_2^*||$ and $||(t_1 \cup t_2)^*|| = 1 + ||t_1^* \cdot t_2^*||$. The argument immediately carries over to the case a proper subterm of t is reduced. □

Now we are ready to present the proof of theorem 1.

Proof (Theorem 1). According to lemma 2, all reduction rules of $\mathcal{R}$ strictly reduce the norms of regexps. Hence, for any $\mathcal{U}$-form regexp t, after at most $||t||$ steps the resulting term cannot be any further reduced. Thus, $\mathcal{R}$ terminates. □

Time complexity of the reduction system (the number of steps performed before the reduction process halts) is quadratic in the *length* of the input regexp. The length of regexp t, denoted $len(t)$, is measured by counting the number of nodes in t's parse tree. The proof of the theorem is straightforward.

Theorem 2. *Time complexity of the reduction system $\mathcal{R}$ is of $\mathcal{O}(len(t)^2)$, where t is the input $\mathcal{U}$-form regexp.*

3.2 Correctness: Preserving Parikh Images

The reduction system $\mathcal{R}$ preserves the Parikh images of regexps.

Theorem 3. *Let $(t;\phi)$ be a constrained regexp, and $(t;\phi) \to (t';\phi')$ by applying one of the rules of $\mathcal{R}$. Then, $\Psi((t;\phi)) = \Psi((t';\phi'))$.*

Proof. Below, we assume that t coincides with the subterm that is reduced. The case a proper subterm of t is reduced follows immediately by structural induction. The proof goes by case analysis.

- Rule (1) is applicable, i.e. $t = (t_1 \cdot t_2)^{*_n}$ and $t' = t_1^{*_n} \cdot t_2^{*_n}$, and $\phi' = \phi$. Suppose e is an evaluation function, and $e(n) = k$ for some k. Note that in general $\Psi(t_1^k \cdot t_2^k) = \Psi((t_1 \cdot t_2)^k)$. Since $\phi' = \phi$, $e \models \phi$ iff $e \models \phi'$. Therefore, $\Psi((t;\phi)) = \Psi((t';\phi'))$.
- Rule (2) is applicable, i.e. $t = (t_1 \cup t_2)^{*_n}$ and $t' = t_1^{*_{n_1}} \cdot t_2^{*_{n_2}}$, and $\phi' = \phi \cup \{n_1 + n_2 = n\}$. Suppose e is an evaluation function, and $e(n) = k$ for some k. In general $\Psi((t_1 \cup t_2)^k) = \bigcup_{(i,j)\in\mathbb{N}\times\mathbb{N},\ i+j=k} \Psi(t_1^i \cdot t_2^j)$. Let e' be an evaluation function equal to e, except that $e'(n_1) = i$ and $e'(n_2) = j$, for some $i, j \in \mathbb{N}$, such that $i + j = k$. Now, if $e \models \phi$, then $e' \models \phi'$. Therefore, $\Psi((t;\phi)) \subseteq \Psi((t';\phi'))$.
  For the other direction, suppose e is any evaluation function such that $e \models \phi'$, that is $e(n_1) = i$, $e(n_2) = j$, $e(n) = k$, for some $i, j, k \in \mathbb{N}$, such that $i+j = k$. Note $e' \models \phi$, simply because n_1 and n_2 are chosen freshly. It is obvious that the language of t'_e is a subset (or equal) to the language of t_e. Therefore, $\Psi((t';\phi')) \subseteq \Psi((t;\phi))$.
- Rule (3) is applicable, that is $t = (t_1^*)^{*_n}$ and $t' = t_1^{*_{n'}}$, and $\phi' = \phi \cup \{n = 0 \implies n' = 0\}$. Suppose e is an evaluation function, and $e(n) = k$ for some k. We consider two cases: (1) $k \neq 0$, and (2) $k = 0$. Case (1): Take an arbitrary $w \in t_e$. Note that in general $(r^*)^k = r^*$, for any regexp r and $k \neq 0$. Therefore, $k \neq 0$ implies $w \in t_1^i$, for some $i \in \mathbb{N}$. Define e' as an evaluation function that is equal to e, except that $e'(n') = i$. Clearly $w \in t'_{e'}$. Since $k \neq 0$, $e \models \phi$ entails $e' \models \phi'$. Case (2): If $k = 0$, then the language of r^k is the set $\{\epsilon\}$ for any regexp r. Define e' as an evaluation function that is equal to e, except that $e'(n') = 0$. Note that the only constraint that is in

ϕ' and not in ϕ (i.e. $n = 0 \implies n' = 0$), is satisfied by e'. Thus, from $e \models \phi$ we derive $e' \models \phi'$. Therefore, $\Psi((t;\phi)) \subseteq \Psi((t';\phi'))$.
For the other direction, suppose e' is any evaluation function such that $e' \models \phi'$, that is $e'(n) = k$, $e'(n') = k'$, for some $k, k' \in \mathbb{N}$, such that $(k \neq 0) \vee (k' = 0)$. It is obvious that $t'_{e'} = t_{e'}$ when $k = 0$ (and consequently $k' = 0$). Now, let $k \neq 0$. Remark that not only n' is freshly chosen from Names and hence not present in t, but also the *-node tagged with t' is not tagged in t at all. This is due to our top-most reduction strategy; see figure 1. This is a crucial fact here, since simply tagging nodes, even when those tags are not bound in any constraint, affects the language of constrained regexps, cf. example 1. Now, from $(r^*)^k = r^*$, with $k \neq 0$, it follows that $\Psi((t';\phi')) \subseteq \Psi((t;\phi))$.

This completes our proof. □

4 Extracting Parikh Images from Most-Reduced Regexps

Suppose $(t_i;\emptyset)$ is a seeded $\mathcal{U}$-form regexp, and $(t_i;\emptyset)$ is reduced using $\mathcal{R}$ to $(t;\phi)$, where t is irreducible. According to theorem 1, such a t is always reached in a finite number of reduction steps, and due to theorem 3, $\Psi(t_i;\emptyset) = \Psi(t;\phi)$. This in particular implies $\Psi(t_i) = \Psi(t;\phi)$, due to the seeding procedure. Our goal here is to extract a semi-linear representations of $\Psi(t_i)$ from $(t;\phi)$. We consider two cases: (i) ϕ contains no implication constraints (i.e. no constraints of the form $n_i = 0 \implies n_j = 0$), and (ii) ϕ contains at least one implication constraint. Computing a semi-linear Parikh image of t_i in case (i) is straightforward, as described below. For case (ii), we give a procedure, called *remove_imp*, which removes the implication constraints from ϕ, and returns a finite set of implication-free constraint sets. It is then proved (theorem 4) that $\Psi(t_i) = \cup_{\varphi \in remove\_imp(\phi)} \Psi(t;\varphi)$. Intuitively, case (ii) is reduced to case (i).

Case (i). Constraint set with no implication constraint. In case ϕ contains no implication constraints, computing semi-linear representations of the Parikh image of $(t;\phi)$ is straightforward. This is because constraints of the form $n_i + n_j = n_k$ are inherently linear constraints. We demonstrate this via a number of examples.

Example 3. Let $t_i = (a \cdot (b \cup c))^{*_n}$, with alphabet $A = \{a, b, c\}$. The following steps show how $\mathcal{R}$ reduces $(t_i;\emptyset)$.

$$\begin{array}{rl} & (a \cdot (b \cup c))^{*_n} \quad ;\emptyset \\ \rightarrow_{R_1} & a^{*_n} \cdot (b \cup c)^{*_n} \quad ;\emptyset \\ \rightarrow_{R_2} & a^{*_n} \cdot b^{*_{n_1}} \cdot c^{*_{n_2}} \; ;\{n_1 + n_2 = n\} \end{array}$$

From the final irreducible constrained regexp we get $\Psi(t_i) = \{v \in \mathbb{N}^3 \mid v = (\lambda_b + \lambda_c)\mathbf{e}_1 + \lambda_b\mathbf{e}_2 + \lambda_c\mathbf{e}_3,\ \lambda_b, \lambda_c \in \mathbb{N}\}$. This is indeed a linear representation. •

Example 4. Let $t_i = ((a \cup b) \cdot (c \cup d) \cdot e)^{*_n}$, with alphabet $A = \{a, b, c, d, e\}$. The following steps show how $\mathcal{R}$ reduces $(t_i; \emptyset)$.

$$\begin{array}{ll} & ((a \cup b) \cdot (c \cup d) \cdot e)^{*_n} \quad ; \emptyset \\ \rightarrow_{R_1} & (a \cup b)^{*_n} \cdot ((c \cup d) \cdot e)^{*_n} \quad ; \emptyset \\ \rightarrow_{R_1} & (a \cup b)^{*_n} \cdot (c \cup d)^{*_n} \cdot e^{*_n} \quad ; \emptyset \\ \rightarrow_{R_2} & a^{*_{n_1}} \cdot b^{*_{n_2}} \cdot (c \cup d)^{*_n} \cdot e^{*_n} \quad ; \{n_1 + n_2 = n\} \\ \rightarrow_{R_2} & a^{*_{n_1}} \cdot b^{*_{n_2}} \cdot c^{*_{n_3}} \cdot d^{*_{n_4}} \cdot e^{*_n} \quad ; \{n_1 + n_2 = n; n_3 + n_4 = n\} \end{array}$$

Notice that equations of the form $n_1 + n_2 = n_3 + n_4$, with $n_i \in \mathbb{N}$, are satisfiable iff there exist $\lambda_1, \lambda_2, \lambda_3, \lambda_4 \in \mathbb{N}$, such that $n_1 = \lambda_1 + \lambda_2$, $n_2 = \lambda_3 + \lambda_4$, $n_3 = \lambda_1 + \lambda_3$ and $n_4 = \lambda_2 + \lambda_4$. In general, any finite number of equations of the form $A_1 + B_1 = \cdots = A_\ell + B_\ell$, with $A_{i \in 1..\ell}, B_{i \in 1..\ell} \in \mathbb{N}$, are simultaneously satisfiable iff there exist 2^ℓ natural numbers $\lambda_1, \cdots, \lambda_{2^\ell}$, whose combinations constitute A_i and B_i. Therefore, from the final irreducible constrained regexp we get $\Psi(t_i) = \{v \in \mathbb{N}^5 \mid v = (\lambda_1 + \lambda_2)\mathbf{e}_1 + (\lambda_3 + \lambda_4)\mathbf{e}_2 + (\lambda_1 + \lambda_3)\mathbf{e}_3 + (\lambda_2 + \lambda_4)\mathbf{e}_4 + (\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4)\mathbf{e}_5,\ \lambda_{i \in 1..4} \in \mathbb{N}\}$. This is indeed a linear representation. •

Case (ii). Constraint set with at least one implication constraint. Implication constraints, i.e. constraints of the form $n_i = 0 \implies n_j = 0$, are intuitively "non-linear". In the following, we give a case splitting procedure to remove such constraints from ϕ, without affecting the set of evaluation functions that satisfy ϕ. This implies that the procedure does not change the language of $(t; \phi)$.

Let ϕ be a set of constraints of the form defined in section 2 over Names. We give an algorithm to remove implication constraints from ϕ. For this purpose, we extend the definition of constraint sets (given in section 2) to include constraints of the form $n = 0$, and $n = n' + 1$ as eligible members. The related definitions (satisfiability, etc.) are extended in the obvious way. Furthermore, in the following we assume that all implication constraints in ϕ which have the same antecedent are lumped together. For example, two constraints $(n = 0 \implies n_1 = 0)$ and $(n = 0 \implies n_2 = 0)$ are lumped into the constraint $n = 0 \implies n_1 = 0 \wedge n_2 = 0$. This merely syntactical convention decreases the number of times the procedure *remove_imp* recurs on ϕ.

Theorem 4. *Fix a constrained regexp $(t; \phi)$. Given the set of constraints ϕ, algorithm 1 returns a finite set of constraint sets $\Phi = \{\phi_1, \cdots, \phi_n\}$ such that $L(t; \phi) = \cup_{\phi_i \in \Phi} L(t; \phi_i)$.*

Proof. Remark that Φ is finite because the number of implication constraints in ϕ is strictly decreasing in each recursion of algorithm 1. The claim of the theorem is immediate by noting that for any evaluation function e, $e \models \phi \iff \bigvee_{\phi_i \in \Phi} e \models \phi_i$. This is because $\forall n \in \mathbb{N}.\ (n = 0 \vee \exists n_\nu \in \mathbb{N}.\ n = n_\nu + 1)$. □

Example 5. Let $\phi = \{n_3 = 0 \implies n_4 = 0; n_2 + n_3 = n; n = 0 \implies n_1 = 0\}$. Suppose the first implication chosen by algorithm 1 is $n_3 = 0 \implies n_4 = 0$. This results in two constraint sets $\phi_1 = \{n_3 = 0; n_4 = 0; n_2 + n_3 = n; n = 0 \implies n_1 = 0\}$ and $\phi_2 = \{n_3 = n' + 1; n_2 + n_3 = n; n = 0 \implies n_1 = 0\}$. Recursively,

Algorithm 1. Removes implications from constraint sets

procedure *remove_imp*(ϕ)
  if there exists a constraint $(n = 0 \implies n_1 = 0 \wedge \cdots \wedge n_\ell = 0)$ in ϕ **then**
    let $\phi := \phi \setminus \{n = 0 \implies n_1 = 0 \wedge \cdots \wedge n_\ell = 0\}$
    return
      remove_imp($\{n = 0, n_1 = 0, \cdots, n_\ell = 0\} \cup \phi$)
        $\cup$
      remove_imp($\{n = n_\nu + 1\} \cup \phi$) where $n_\nu \in \mathsf{Names}$ is freshly chosen
  end if
  return $\{\phi\}$

on ϕ_1 the algorithm returns $\phi_3 = \{n_3 = 0; n_4 = 0; n_2 + n_3 = n; n = 0; n_1 = 0\}$ and $\phi_4 = \{n_3 = 0; n_4 = 0; n_2 + n_3 = n; n = n'' + 1\}$. Similarly, on ϕ_2 the algorithm returns $\phi_5 = \{n_3 = n' + 1; n_2 + n_3 = n; n = 0; n_1 = 0\}$ and $\phi_6 = \{n_3 = n' + 1; n_2 + n_3 = n; n = n'' + 1\}$. Ultimately, *remove_imp*$(\phi) = \{\phi_3, \phi_4, \phi_5, \phi_6\}$. •

Algorithm 1 effectively transforms constraint set ϕ into a finite set of constraint sets $\Phi = \{\phi_1 \cdots, \phi_\ell\}$, such that ϕ_i contains no implication constraints. That is, case (ii) is reduced to case (i), except for constraints of the form $n = n_\nu + 1$ that algorithm 1 may introduce. To cast constraints of the form $n = n_\nu + 1$ into case (i) we need another round of case distinction. This is best explained via an example.

Example 6. Let $t_i = ((a \cup b) \cdot c^* \cdot d)^*$, with $A = \{a, b, c, d\}$. The most-reduced constrained regexp that $\mathcal{R}$ produces from t_i is then $(t; \phi)$ with $t = a^{*_{n_1}} \cdot b^{*_{n_2}} \cdot c^{*_{n_3}} \cdot d^{*_n}$ and $\phi = \{n = 0 \implies n_3 = 0, n_1 + n_2 = n\}$. Then, *remove_imp*$(\phi) = \{\phi_1, \phi_2\}$ where $\phi_1 = \{n = 0, n_3 = 0, n_1 + n_2 = n\}$ and $\phi_2 = \{n = n_\nu + 1, n_1 + n_2 = n\}$. The constraints in ϕ_1 are all linear, hence fall into case (i). In ϕ_2 however the constraint $n_1 + n_2 = n_\nu + 1$ results in $n_1 = \lambda_1 + \lambda_2$, $n_2 = \lambda_3 + \lambda_4$, $n_\nu = \lambda_1 + \lambda_3$ and $1 = \lambda_2 + \lambda_4$ (cf. example 4). The last constraint needs a case distinction; namely, either $\lambda_2 = 1$ and $\lambda_4 = 0$, or $\lambda_2 = 0$ and $\lambda_4 = 1$. With this case distinction, ϕ_2 also falls into case (i). •

Width of produced images. We now turn to the width of produced semi-linear representations of Parikh images of regexps.

Theorem 5. *Let t be a $\mathcal{U}$-form regexp. Then, the width of $\Psi(t)$, produced by the reduction process, is (at worst case) given by*

$$\omega(\Psi(t)) = 2^{\aleph_2(t) + \aleph_3(t)}$$

Proof. Take seeded constrained regexp $(t; \emptyset)$ and suppose it is reduced using $\mathcal{R}$ to $(r; \phi)$, where r is irreducible. The key observation is that the width of $\Psi(t)$ is equal to $|$*remove_imp*$(\phi)|$ multiplied by the number of case distinctions performed for linearizing $n = n_\nu + 1$ constraints.

Note that $|$*remove_imp*$(\phi)| = 2^{\aleph_3(t)}$, by definition 2. Now, all the sets in *remove_imp*(ϕ) (might) require further case distinctions for constraints of the

form $n = n_\nu + 1$ (cf. example 6). The number of these case distinctions is $2^{\aleph_2(t)}$, according to definition 2. The number of linear components in $\Psi(t)$ is therefore bounded by $2^{\aleph_2(t)+\aleph_3(t)}$. □

To relate theorem 5 to the measures given in the introduction (section 1), let us assume that the widths of images of $\mathcal{U}$-form regexps t_1 and t_2 are, respectively, w_1 and w_2. The reduction process generates an image of width $w_1 + w_2$ for $t_1 \cup t_2$ (cf. lemma 1). For $t_1 \cdot t_2$, we note that $\aleph_i(t_1 \cdot t_2) = \aleph_i(t_1) + \aleph_i(t_2)$, for $i \in \{2, 3\}$. Then, due to theorem 5, $\omega(\Psi(t_1 \cdot t_2)) = w_1 w_2$. For t_1^*, we remark that reducing t_1^* using $\mathcal{R}$ results in a constraint set which has at most one implication constraint more than the constraint set generated for t_1. This is due to lumping the implication constraints (see the discussion right before theorem 4). Then, due to theorem 5, $\omega(\Psi(t_1^*)) = 2^{1+\aleph_2(t_1)+\aleph_3(t_1)}$, that is $\omega(\Psi(t_1^*)) = 2w_1$.

Acknowledgement. We are grateful to B. Conchinha Montalto, W. Fokkink and L. Schrijver for fruitful discussions. Mohammad Torabi Dashti has been supported by AVANTSSAR, FP7-ICT-2007-1 Project no. 216471.

References

1. Aceto, L., Ésik, Z., Ingólfsdóttir, A.: A fully equational proof of Parikh's theorem. ITA 36(2), 129–153 (2002)
2. Blattner, M., Latteux, M.: Parikh-bounded languages. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 316–323. Springer, Heidelberg (1981)
3. Delgado, M.: Commutative images of rational languages and the Abelian kernel of a monoid. ITA 35(5), 419–435 (2001)
4. Goldstine, J.: A simplified proof of Parikh's theorem. Discrete Math. 19, 235–239 (1977)
5. Ibarra, O., Kim, C.: A useful device for showing the solvability of some decision problems. In: STOC 1976, pp. 135–140. ACM, New York (1976)
6. Parikh, R.: On context-free languages. J. ACM 13(4), 570–581 (1966)
7. Pilling, D.: Commutative regular equations and Parikh's theorem. J. London Math. Soc. 6, 633–666 (1973)
8. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
9. Verma, K.N., Seidl, H., Schwentick, T.: On the complexity of equational Horn clauses. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 337–352. Springer, Heidelberg (2005)
10. Xie, G., Li, C., Deng, Z.: Linear reachability problems and minimal solutions to linear Diophantine equation systems. TCS 328, 203–219 (2004)