Some annotation
and also this one

Computing Downward Closures for Stacked Counter Automata

Georg Zetzsche

AG Concurrency Theory
Fachbereich Informatik
TU Kaiserslautern
zetzsche@cs.uni-kl.de

Abstract

The downward closure of a language L of words is the set of all (not necessarily contiguous) subwords of members of L. It is well known that the downward closure of any language is regular. Although the downward closure seems to be a promising abstraction, there are only few language classes for which an automaton for the downward closure is known to be computable.

It is shown here that for stacked counter automata, the downward closure is computable. Stacked counter automata are finite automata with a storage mechanism obtained by *adding blind counters* and *building stacks*. Hence, they generalize pushdown and blind counter automata.

The class of languages accepted by these automata are precisely those in the hierarchy obtained from the context-free languages by alternating two closure operators: imposing semilinear constraints and taking the algebraic extension. The main tool for computing downward closures is the new concept of Parikh annotations. As a second application of Parikh annotations, it is shown that the hierarchy above is strict at every level.

1998 ACM Subject Classification F.4.3 Formal languages

Keywords and phrases abstraction, downward closure, obstruction set, computability

Digital Object Identifier 10.4230/LIPIcs.STACS.2015.743

1 Introduction

In the analysis of systems whose behavior is given by formal languages, it is a fruitful idea to consider abstractions: simpler objects that preserve relevant properties of the language and are amenable to algorithmic examination. A well-known such type of abstraction is the *Parikh image*, which counts the number of occurrences of each letter. For a variety of language classes, the Parikh image of every language is known to be effectively semilinear, which facilitates a range of analysis techniques for formal languages (see [12] for applications).

A promising alternative to Parikh images is the *downward closure* $L\downarrow$, which consists of all (not necessarily contiguous) subwords of members of L. Whereas for many interesting classes of languages the Parikh image is not semilinear in general, the downward closure is regular *for any language* [10], suggesting wide applicability. Moreover, the downward closure encodes properties not visible in the Parikh image: Suppose L describes the behavior of a system that is observed through a lossy channel, meaning that on the way to the observer, arbitrary actions can get lost. Then, $L\downarrow$ is the set of words received by the observer [9]. Hence, given the downward closure as a finite automaton, we can decide whether two systems are equivalent under such observations, and even whether the behavior of one system includes the other. Hence, even if Parikh images are effectively semilinear for a class of languages, computing the downward closure is still an important task. See [2, 3, 13] for further applications.

32nd Symposium on Theoretical Aspects of Computer Science (STACS 2015).
Editors: Ernst W. Mayr and Nicolas Ollinger; pp. 743–756
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

However, while there always *exists* a finite automaton for the downward closure, it seems difficult to *compute* them and there are few language classes for which computability has been established. The downward closure is known to be computable for context-free languages and algebraic extensions [5, 16], 0L-systems and context-free FIFO rewriting systems [1], and Petri net languages [9]. It is not computable for reachability sets of lossy channel systems [14] and for Church-Rosser languages [7]. For considerations of complexity, both descriptional and computational, see [3, 8, 11, 15] and the references therein.

It is shown here that downward closures are computable for *stacked counter automata*. These are automata with a finite state control and a storage mechanism obtained by two constructions (of storage mechanisms): One can *build stacks* and *add blind counters*. The former is to construct a new mechanism that stores a stack whose entries are configurations of an old mechanism. One can then manipulate the topmost entry, pop it if empty, or start a new one on top. Adding a blind counter to an old mechanism yields a new mechanism in which the old one and a blind counter (i.e., a counter that can attain negative values and has to be zero in the end of a run) can be used simultaneously.

Stacked counter automata are interesting because among a large class of automata with storage, they are *expressively complete* for those storage mechanisms that guarantee semilinear Parikh images. This is due to the fact that they accept precisely those languages in the hierarchy obtained from the context-free languages by alternating two closure operators: imposing semilinear constraints (with respect to the Parikh image) and taking the algebraic extension. These two closure operators correspond to the constructions of storage mechanisms in stacked counter automata (see Section 3).

The main tool to show the computability of downward closures is the concept of *Parikh annotations*. As another application of this concept, it is shown that the aforementioned hierarchy is strict at every level.

The paper is structured as follows. After Section 2 defines basic concepts and notation, Section 3 introduces the hierarchy of language classes. Section 4 presents Parikh annotations, the main ingredient for the computation of downward closures. The main result is then presented in Section 5, where it is shown that downward closures are computable for stacked counter automata. As a second application of Parikh annotations, it is then shown in Section 6 that the hierarchy defined in Section 3 is strict at every level. Because of space restrictions, most proofs can only be found in the long version of this work [18].

2 Preliminaries

A *monoid* is a set M together with a binary associative operation such that M contains a neutral element. Unless the monoid at hand warrants a different notation, we will denote the neutral element by 1 and the product of $x, y \in M$ by xy. The trivial monoid that contains only the neutral element is denoted by $\mathbf{1}$.

If X is an alphabet, X^* denoted the set of words over X. The empty word is denoted by $\varepsilon \in X^*$. For a symbol $x \in X$ and a word $w \in X^*$, let $|w|_x$ be the number of occurrences of x in w and $|w| = \sum_{x\in X} |w|_x$. For an alphabet X and languages $L, K \subseteq X^*$, the *shuffle product* $L \sqcup\!\sqcup K$ is the set of all words $u_0v_1u_1 \cdots v_nu_n$ where $u_0, \ldots, u_n, v_1, \ldots, v_n \in X^*$, $u_0 \cdots u_n \in L$, and $v_1 \cdots v_n \in K$. For a subset $Y \subseteq X$, we define the *projection morphism* $\pi_Y : X^* \to Y^*$ by $\pi_Y(y) = y$ for $y \in Y$ and $\pi_Y(x) = \varepsilon$ for $x \in X \setminus Y$. By $\mathcal{P}(S)$, we denote the power set of the set S. A *substitution* is a map $\sigma : X \to \mathcal{P}(Y^*)$ and given $L \subseteq X^*$, we write $\sigma(L)$ for the set of all words $v_1 \cdots v_n$, where $v_i \in \sigma(x_i)$, $1 \le i \le n$, for $x_1 \cdots x_n \in L$ and $x_1, \ldots, x_n \in X$. If $\sigma(x) \subseteq Y$ for each $x \in X$, we call σ a *letter substitution*.

For words $u, v \in X^*$, we write $u \preceq v$ if $u = u_1 \cdots u_n$ and $v = v_0u_1v_1 \cdots u_nv_n$ for some $u_1, \ldots, u_n, v_0, \ldots, v_n \in X^*$. It is well-known that $\preceq$ is a well-quasi-order on X^* and that therefore the *downward closure* $L{\downarrow} = \{u \in X^* \mid \exists v \in L\colon u \preceq v\}$ is regular for any $L \subseteq X^*$ [10].

If X is an alphabet, $X^\oplus$ denotes the set of maps $\alpha\colon X \to \mathbb{N}$. The elements of $X^\oplus$ are called *multisets*. Let $\alpha + \beta \in X^\oplus$ be defined by $(\alpha + \beta)(x) = \alpha(x) + \beta(x)$. With this operation, $X^\oplus$ is a monoid. We consider each $x \in X$ to be an element of $X^\oplus$. For a subset $S \subseteq X^\oplus$, we write $S^\oplus$ for the smallest submonoid of $X^\oplus$ containting S. For $\alpha \in X^\oplus$ and $k \in \mathbb{N}$, we define $(k \cdot \alpha)(x) = k \cdot \alpha(x)$, meaning $k \cdot \alpha \in X^\oplus$. A subset of the form $\mu + F^\oplus$ for $\mu \in X^\oplus$ and a finite $F \subseteq X^\oplus$ is called *linear*. A finite union of linear sets is called *semilinear*. The *Parikh map* is the map $\Psi\colon X^* \to X^\oplus$ defined by $\Psi(w)(x) = |w|_x$ for all $w \in X^*$ and $x \in X$. Given a morphism $\varphi\colon X^\oplus \to Y^\oplus$ and a word $w \in X^*$, we use $\varphi(w)$ as a shorthand for $\varphi(\Psi(w))$. We lift Ψ to sets in the usual way: $\Psi(L) = \{\Psi(w) \mid w \in L\}$. If $\Psi(L)$ is semilinear, we will also call L itself semilinear.

Let M be a monoid. An *automaton over* M is a tuple $A = (Q, M, E, q_0, F)$, in which

(i) Q is a finite set of *states*,
(ii) E is a finite subset of $Q \times M \times Q$ called the set of *edges*,
(iii) $q_0 \in Q$ is the *initial state*, and
(iv) $F \subseteq Q$ is the set of *final states*.

We write $(q, m) \to_A (q', m')$ if there is an edge $(q, r, q') \in E$ such that $m' = mr$. The set *generated* by A is then $S(A) = \{m \in M \mid (q_0, 1) \to_A^* (f, m) \text{ for some } f \in F\}$.

A *class of languages* is a collection of languages that contains at least one non-empty language. The class of regular languages is denoted by REG. A *finite state transducer* is an automaton over $Y^* \times X^*$ for alphabets X, Y. Relations of the form $S(A)$ for finite state transducers A are called *rational transductions*. For $L \subseteq X^*$ and $T \subseteq Y^* \times X^*$, we write $TL = \{u \in Y^* \mid \exists v \in L\colon (u, v) \in T\}$. If TF is finite for every finite language F, T is said to be *locally finite*. A class $\mathcal{C}$ of languages is called a *full trio* (*full semi-trio*) if it is closed under (locally finite) rational transductions, i.e. if $TL \in \mathcal{C}$ for every $L \in \mathcal{C}$ and every (locally finite) rational transduction T. A *full semi-AFL* is a union closed full trio.

Stacked counter automata In order to define stacked counter automata, we use the concept of valence automata, which combine a finite state control with a storage mechanism defined by a monoid M. A *valence automaton over* M is an automaton A over $X^* \times M$ for an alphabet X. The *language accepted by* A is then $\mathsf{L}(A) = \{w \in X^* \mid (w, 1) \in S(A)\}$. The class of languages accepted by valence automata over M is denoted $\mathsf{VA}(M)$. By choosing suitable monoids M, one can obtain various kinds of automata with storage as valence automata. For example, blind counters, partially blind counters, pushdown storages, and combinations thereof can all be realized by appropriate monoids [19].

If one storage mechanism is realized by a monoid M, then the mechanism that *builds stacks* is realized by the monoid $\mathbb{B} * M$. Here, $\mathbb{B}$ denotes the bicyclic monoid, presented by $\langle a, \bar{a} \mid a\bar{a} = 1\rangle$, and $*$ denotes the free product of monoids. For readers not familiar with these concepts, it will suffice to know that a configuration of the storage mechanism described by $\mathbb{B} * M$ consists of a sequence $c_0ac_1 \cdots ac_n$, where $c_0, \ldots, c_n$ are configurations of the mechanism realized by M. We interpret this as a stack with the entries $c_0, \ldots, c_n$. One can open a new stack entry on top (by multiplying $a \in \mathbb{B}$), remove the topmost entry if empty (by multiplying $\bar{a} \in \mathbb{B}$) and operate on the topmost entry using the old mechanism (by multiplying elements from M). For example, the monoid $\mathbb{B}$ describes a partially blind counter (i.e. a counter that cannot go below zero and is only tested for zero in the end) and

$\mathbb{B} * \mathbb{B}$ describes a pushdown with two stack symbols. Given a storage mechanism realized by a monoid M, we can *add a blind counter* by using the monoid $M \times \mathbb{Z}$, where $\mathbb{Z}$ denotes the group of integers. We define SC to be the smallest class of monoids with $\mathbf{1} \in \mathsf{SC}$ such that whenever $M \in \mathsf{SC}$, we also have $M \times \mathbb{Z} \in \mathsf{SC}$ and $\mathbb{B} * M \in \mathsf{SC}$. A *stacked counter automaton* is a valence automaton over M for some $M \in \mathsf{SC}$. For more details, see [19]. In Section 3, we will turn to a different description of the languages accepted by stacked counter automata.

3 A hierarchy of language classes

This section introduces a hierarchy of language classes that divides the class of languages accepted by stacked counter automata into levels. This will allow us to apply recursion with respect to these levels. The hierarchy is defined by alternating two operators on language classes, algebraic extensions and semilinear intersections.

Algebraic extensions Let $\mathcal{C}$ be a class of languages. A *$\mathcal{C}$-grammar* is a quadruple $G = (N, T, P, S)$ where N and T are disjoint alphabets and $S \in N$. The symbols in N and T are called the *nonterminals* and the *terminals*, respectively. P is a finite set of pairs (A, M) with $A \in N$ and $M \subseteq (N \cup T)^*$, $M \in \mathcal{C}$. A pair $(A, M) \in P$ is called a *production of* G and also denoted by $A \to M$. The set M is the *right-hand side* of the production $A \to M$.

We write $x \Rightarrow_G y$ if $x = uAv$ and $y = uwv$ for some $u, v, w \in (N \cup T)^*$ and $(A, M) \in P$ with $w \in M$. A word w with $S \Rightarrow_G^* w$ is called a *sentential form* of G and we write $\mathsf{SF}(G)$ for the set of sentential forms of G. The *language generated by* G is $\mathsf{L}(G) = \mathsf{SF}(G) \cap T^*$. Languages generated by $\mathcal{C}$-grammars are called *algebraic over* $\mathcal{C}$. The class of all languages that are algebraic over $\mathcal{C}$ is called the *algebraic extension* of $\mathcal{C}$ and denoted $\mathsf{Alg}(\mathcal{C})$. We say a language class $\mathcal{C}$ is *algebraically closed* if $\mathsf{Alg}(\mathcal{C}) = \mathcal{C}$. If $\mathcal{C}$ is the class of finite languages, $\mathcal{C}$-grammars are also called *context-free grammars*.

We will use the operator $\mathsf{Alg}(\cdot)$ to describe the effect of *building stacks* on the accepted languages of valence automata. In [19], it was shown that $\mathsf{VA}(M_0 * M_1) \subseteq \mathsf{Alg}(\mathsf{VA}(M_0) \cup \mathsf{VA}(M_1))$. Here, we complement this by showing that if one of the factors is $\mathbb{B} * \mathbb{B}$, the inclusion becomes an equality. Observe that since $\mathsf{VA}(\mathbb{B} * \mathbb{B})$ is the class of languages accepted by pushdown automata and $\mathsf{Alg}(\mathsf{REG}) = \mathsf{Alg}(\mathsf{VA}(\mathbf{1}))$ is clearly the class of languages generated by context-free grammars, the first statement of the following theorem generalizes the equivalence between pushdown automata and context-free grammars.

▶ **Theorem 1.** *For every monoid M, $\mathsf{Alg}(\mathsf{VA}(M)) = \mathsf{VA}(\mathbb{B} * \mathbb{B} * M)$.*

Semilinear intersections The second operator on language classes lets us describe the languages in $\mathsf{VA}(M \times \mathbb{Z}^n)$ in terms of those in $\mathsf{VA}(M)$. Consider a language class $\mathcal{C}$. By $\mathsf{SLI}(\mathcal{C})$, we denote the class of languages of the form $h(L \cap \Psi^{-1}(S))$, where $L \subseteq X^*$ is in $\mathcal{C}$, the set $S \subseteq X^{\oplus}$ is semilinear, and $h\colon X^* \to Y^*$ is a morphism. We call a language class $\mathcal{C}$ *Presburger closed* if $\mathsf{SLI}(\mathcal{C}) = \mathcal{C}$. Proving the following requires only standard techniques.

▶ **Proposition 2.** *Let M be a monoid. Then $\mathsf{SLI}(\mathsf{VA}(M)) = \bigcup_{n \geq 0} \mathsf{VA}(M \times \mathbb{Z}^n)$.*

The hierarchy is now obtained by alternating the operators $\mathsf{Alg}(\cdot)$ and $\mathsf{SLI}(\cdot)$. Let F_0 be the class of finite languages and let

$$\mathsf{G}_i = \mathsf{Alg}(\mathsf{F}_i), \quad \mathsf{F}_{i+1} = \mathsf{SLI}(\mathsf{G}_i) \quad \text{for each } i \geq 0, \qquad \mathsf{F} = \bigcup_{i \geq 0} \mathsf{F}_i.$$

Then we clearly have the inclusions $\mathsf{F}_0 \subseteq \mathsf{G}_0 \subseteq \mathsf{F}_1 \subseteq \mathsf{G}_1 \subseteq \cdots$. Furthermore, G_0 is the class of context-free languages, F_1 is the smallest Presburger closed class containing CF, G_1 the algebraic extension of F_1, etc. In particular, F is the smallest Presburger closed and algebraically closed language class containing the context-free languages.

The following proposition is due to the fact that both $\mathsf{Alg}(\cdot)$ and $\mathsf{SLI}(\cdot)$ preserve (effective) semilinearity. The former has been shown by van Leeuwen [16].

▶ **Proposition 3.** *The class* F *is effectively semilinear.*

The work [4] characterized all those storage mechanisms among a large class (namely among those defined by graph products of the bicyclic monoid and the integers) that guarantee semilinear Parikh images. Each of the corresponding language classes was obtained by alternating the operators $\mathsf{Alg}(\cdot)$ and $\mathsf{SLI}(\cdot)$, meaning that all these classes are contained in F. Hence, the following means that stacked counter automata are expressively complete for these storage mechanisms. It follows directly from Theorem 1 and Proposition 2.

▶ **Theorem 4.** *Stacked counter automata accept precisely the languages in* F.

One might wonder why F_0 is not chosen to be the regular languages. While this would be a natural choice, our recursive algorithm for computing downward closures relies on the following fact . Note that the regular languages are not Presburger closed.

▶ **Proposition 5.** *For each $i \geq 0$, the class F_i is an effective Presburger closed full semi-trio. Moreover, for each $i \geq 0$, G_i is an effective full semi-AFL.*

4 Parikh annotations

This section introduces Parikh annotations, the key tool in our procedure for computing downward closures. Suppose L is a semilinear language. Then for each $w \in L$, $\Psi(w)$ can be decomposed into a constant vector and a linear combination of period vectors from the semilinear representation of $\Psi(L)$. We call such a decomposition a *Parikh decomposition*. The main purpose of Parikh annotations is to provide transformations of languages that *make reference to Parikh decompositions* without leaving the respective language class. For example, suppose we want to transform a context-free language L into the language L' of all those words $w \in L$ whose Parikh decomposition does not contain a specified period vector. This may not be possible with rational transductions: If $L_\vee = \{a^n b^m \mid m = n \text{ or } m = 2n\}$, then the Parikh image is $(a+b)^\oplus \cup (a+2b)^\oplus$, but a finite state transducer cannot determine whether the input word has a Parikh image in $(a+b)^\oplus$ or in $(a+2b)^\oplus$. Therefore, a Parikh annotation for L is a language K in the same class with additional symbols that allow a finite state transducer (that is applied to K) to access the Parikh decomposition.

▶ **Definition 6.** *Let $L \subseteq X^*$ be a language and $\mathcal{C}$ be a language class. A* Parikh annotation (PA) for *L* in *$\mathcal{C}$ is a tuple $(K, C, P, (P_c)_{c\in C}, \varphi)$, where*
1. *C, P are alphabets such that X, C, P are pairwise disjoint,*
2. *$K \subseteq C(X \cup P)^*$ is in $\mathcal{C}$,*
3. *φ is a morphism $\varphi\colon (C \cup P)^\oplus \to X^\oplus$,*
4. *P_c is a subset $P_c \subseteq P$ for each $c \in C$,*

such that

(i) $\pi_X(K) = L$ *(the* projection property*),*
(ii) $\varphi(\pi_{C\cup P}(w)) = \Psi(\pi_X(w))$ *for each $w \in K$ (the* counting property*), and*
(iii) $\Psi(\pi_{C\cup P}(K)) = \bigcup_{c\in C} c + P_c^\oplus$ *(the* commutative projection property*).*

A Parikh annotation describes for each w in L one or more Parikh decompositions of $\Psi(w)$. The symbols in C represent constant vectors and symbols in P represent period vectors. The symbols in $P_c \subseteq P$ correspond to those that can be added to the constant vector corresponding to $c \in C$. Furthermore, for each $x \in C \cup P$, $\varphi(x)$ is the vector represented by x. The projection property states that removing the symbols in $C \cup P$ from words in K yields L. The commutative projection property requires that after $c \in C$ only symbols representing periods in P_c are allowed and that all their combinations occur. Finally, the counting property says that the additional symbols in $C \cup P$ indeed describe a Parikh decomposition of $\Psi(\pi_X(w))$. Of course, only semilinear languages can have a Parikh annotations.

▶ **Example 7.** *Let $X = \{a, b, c, d\}$ and consider the regular set $L = (ab)^*(ca^* \cup db^*)$. For $K = e(pab)^*c(qa)^* \cup f(rab)^*d(sb)^*$, $P = \{p, q, r, s\}$, $C = \{e, f\}$, $P_e = \{p, q\}$, $P_f = \{r, s\}$, and $\varphi\colon (C \cup P)^{\oplus} \to X^{\oplus}$ with $e \mapsto c$, $f \mapsto d$, $p \mapsto a + b$, $q \mapsto a$, $r \mapsto a + b$, and $s \mapsto b$, the tuple $(K, C, P, (P_g)_{g \in C}, \varphi)$ is a Parikh annotation for L in* REG.

In a Parikh annotation, for each $cw \in K$ and $\mu \in P_c^{\oplus}$, we can find a word cw' in K such that $\Psi(\pi_{C \cup P}(cw')) = \Psi(\pi_{C \cup P}(cw)) + \mu$. In particular, this implies the equality $\Psi(\pi_X(cw')) = \Psi(\pi_X(cw)) + \varphi(\mu)$. In our applications, we will need a further guarantee that provides such words, but with additional information on their structure. Such a guarantee is granted by Parikh annotations with insertion marker. Suppose $\diamond \notin X$ and $u \in (X \cup \{\diamond\})^*$ with $u = u_0 \diamond u_1 \cdots \diamond u_n$ for $u_0, \ldots, u_n \in X^*$. Then we write $u \preceq_{\diamond} v$ if $v = u_0 v_1 u_1 \cdots v_n u_n$ for some $v_1, \ldots, v_n \in X^*$.

▶ **Definition 8.** *Let $L \subseteq X^*$ be a language and $\mathcal{C}$ be a language class. A* Parikh annotation with insertion marker (PAIM) for L in $\mathcal{C}$ *is a tuple $(K, C, P, (P_c)_{c \in C}, \varphi, \diamond)$ such that:*

(i) *$\diamond \notin X$ and $K \subseteq C(X \cup P \cup \{\diamond\})^*$ is in $\mathcal{C}$,*
(ii) *$(\pi_{C \cup X \cup P}(K), C, P, (P_c)_{c \in C}, \varphi)$ is a Parikh annotation for L in $\mathcal{C}$,*
(iii) *there is a $k \in \mathbb{N}$ such that every $w \in K$ satisfies $|w|_{\diamond} \le k$ (*boundedness*), and*
(iv) *for each $cw \in K$ and $\mu \in P_c^{\oplus}$, there is a $w' \in L$ with $\pi_{X \cup \diamond}(cw) \preceq_{\diamond} w'$ and with $\Psi(w') = \Psi(\pi_X(cw)) + \varphi(\mu)$. This property is called the* insertion property.

If $|C| = 1$, then the PAIM is called linear *and we also write $(K, c, P_c, \varphi, \diamond)$ for the PAIM, where $C = \{c\}$.*

In other words, in a PAIM, each $v \in L$ has an annotation $cw \in K$ in which a bounded number of positions is marked such that for each $\mu \in P_c^{\oplus}$, we can find a $v' \in L$ with $\Psi(v') = \Psi(v) + \varphi(\mu)$ such that v' is obtained from v by inserting words in corresponding positions in v. In particular, this guarantees $v \preceq v'$.

▶ **Example 9.** *Let L and $(K, C, P, (P_c)_{c \in C}, \varphi)$ be as in Example 7. Furthermore, let $K' = e \diamond (pab)^*c \diamond (qa)^* \cup f \diamond (rab)^*d \diamond (sb)^*$. Then $(K', C, P, (P_c)_{c \in C}, \varphi, \diamond)$ is a PAIM for L in* REG*. Indeed, every word in K' has at most two occurrences of $\diamond$. Moreover, if $ew = e \diamond (pab)^m c \diamond (qa)^n \in K'$ and $\mu \in P_e^{\oplus}$, $\mu = k \cdot p + \ell \cdot q$, then $w' = (ab)^{k+m}ca^{\ell+n} \in L$ satisfies $\pi_{X \cup \diamond}(ew) = \diamond(ab)^m c \diamond a^n \preceq_{\diamond} (ab)^k(ab)^m ca^{\ell}a^n = w'$ and clearly $\Psi(\pi_X(w')) = \Psi(\pi_X(ew)) + \varphi(\mu)$ (and similarly for words $fw \in K'$).*

The main result of this section is that there is an algorithm that, given a language $L \in \mathsf{F}_i$ or $L \in \mathsf{G}_i$, constructs a PAIM for L in F_i or G_i, respectively.

▶ **Theorem 10.** *Given $i \in \mathbb{N}$ and L in F_i (G_i), one can construct a PAIM for L in F_i (G_i).*

Outline of the proof The rest of this section is devoted to the proof of Theorem 10. The construction of PAIM proceeds recursively with respect to the level of our hierarchy. This means, we show that if PAIM can be constructed for F_i, then we can compute them for G_i (Lemma 17) and if they can be constructed for G_i, then they can be computed for F_{i+1} (Lemma 18). While the latter can be done with a direct construction, the former requires a series of involved steps:

- The general idea is to use recursion with respect to the number of nonterminals: Given a F_i-grammar for $L \in \mathsf{G}_i$, we present L in terms of languages whose grammars use fewer nonterminals. This presentation is done via substitutions and by using grammars with one nonterminal. The idea of presenting a language in $\mathsf{Alg}(\mathcal{C})$ using one-nonterminal grammars and substitutions follows van Leeuwen's proof of Parikh's theorem [16].
- We construct PAIM for languages generated by one-nonterminal grammars where we are given PAIM for the right-hand-sides (Lemma 16).
- We construct PAIM for languages $\sigma(L)$, where σ is a substitution, a PAIM is given for L and for each $\sigma(x)$ (Lemma 15). This construction is again divided into the case where σ is a letter substitution (i.e., one in which each symbol is mapped to a set of letters) and the general case. Since the case of letter substitutions constitutes the conceptually most involved step, part of its proof is contained in this extended abstract (Proposition 13).

Maybe surprisingly, the most conceptually involved step in the construction of PAIM lies within obtaining a Parikh annotation for $\sigma(L)$ in $\mathsf{Alg}(\mathcal{C})$, where σ is a letter substitution and a PAIM for $L \subseteq X^*$ in $\mathsf{Alg}(\mathcal{C})$ is given. This is due to the fact that one has to substitute the symbols in X consistently with the symbols in $C \cup P$; more precisely, one has to maintain the agreement between $\varphi(\pi_{C\cup P}(\cdot))$ and $\Psi(\pi_X(\cdot))$.

In order to exploit the fact that this agreement exists in the first place, we use the following simple yet very useful lemma. It states that for a morphism ψ into a group, the only way a grammar G can guarantee $\mathsf{L}(G) \subseteq \psi^{-1}(h)$ is by encoding into each nonterminal A the value $\psi(u)$ for the words u that A derives. The G-compatible extension of ψ reconstructs this value for each nonterminal. Let $G = (N, T, P, S)$ be a $\mathcal{C}$-grammar and M be a monoid. A morphism $\psi\colon (N \cup T)^* \to M$ is called *G-compatible* if $u \Rightarrow_G^* v$ implies $\psi(u) = \psi(v)$ for $u, v \in (N \cup T)^*$. Moreover, we call G *reduced* if for each $A \in N$, we have $A \Rightarrow_G^* w$ for some $w \in T^*$ and $S \Rightarrow_G^* uAv$ for some $u, v \in (N \cup T)^*$.

▶ **Lemma 11.** *Let H be a group, $\psi\colon T^* \to H$ be a morphism, and $G = (N, T, P, S)$ be a reduced $\mathcal{C}$-grammar with $\mathsf{L}(G) \subseteq \psi^{-1}(h)$ for some $h \in H$. Then ψ has a unique G-compatible extension $\hat{\psi}\colon (N \cup T)^* \to H$. If $H = \mathbb{Z}^n$ and $\mathcal{C} = \mathsf{F}_i$, $\hat{\psi}$ can be computed.*

We will essentially apply Lemma 11 by regarding $X^\oplus$ as a subset of $\mathbb{Z}^n$ and defining $\psi\colon (C \cup P \cup X)^* \to \mathbb{Z}^n$ as the morphism with $\psi(w) = \Psi(\pi_X(w)) - \varphi(\pi_{C\cup P}(w))$. In the case that G generates the corresponding Parikh annotation, the counting property implies that $\mathsf{L}(G) \subseteq \psi^{-1}(0)$. The lemma then states that each nonterminal in G encodes the imbalance between $\Psi(\pi_X(\cdot))$ and $\varphi(\pi_{C\cup P}(\cdot))$ on the words it generates.

We continue with the problem of replacing $C \cup P$ and X consistently. For constructing the PAIM for $\sigma(L)$, it is easy to see that it suffices to consider the case where $\sigma(a) = \{a, b\}$ for some $a \in X$ and $\sigma(x) = \{x\}$ for $x \in X \setminus \{a\}$. In order to simplify the setting and exploit the symmetry of the roles played by $C \cup P$ and X, we consider a slightly more general situation. There is an alphabet $X = X_0 \uplus X_1$, morphisms $\gamma_i\colon X_i^* \to \mathbb{N}$, $i = 0, 1$, and a language $L \subseteq X^*$, $L \in \mathsf{Alg}(\mathsf{F}_i)$ with $\gamma_0(\pi_{X_0}(w)) = \gamma_1(\pi_{X_1}(w))$ for every $w \in L$. Roughly speaking, X_1 will later play the role of $C \cup P$ and X_0 will play the role of X. Then, $\gamma_0(w)$

will be the number of a's in w and $\gamma_1(w)$ will be the number of a's represented by symbols from $C \cup P$ in w. Therefore, we wish to construct a language L' in $\mathsf{Alg}(\mathsf{F}_i)$ such that each word in L' is obtained from a word in L as follows. We substitute each occurrence of $x \in X_i$ by one of $\gamma_i(x)+1$ many symbols y in an alphabet Y_i, each of which will be assigned a value $0 \le \eta_i(y) \le \gamma_i(x)$. Here, we want to guarantee that in every resulting word $w \in (Y_0 \cup Y_1)^*$, we have $\eta_0(\pi_{Y_0}(w)) = \eta_1(\pi_{Y_1}(w))$, meaning that the symbols in X_0 and in X_1 are replaced *consistently.* Formally, we have

$$Y_i = \{(x,j) \mid x \in X_i,\ 0 \le j \le \gamma_i(x)\}, \quad i = 0,1, \qquad Y = Y_0 \cup Y_1, \tag{1}$$

and the morphisms

$$\begin{aligned} h_i\colon Y_i^* &\longrightarrow X_i^*, & h\colon Y^* &\longrightarrow X^*, & \eta_i\colon Y_i^* &\longrightarrow \mathbb{N}, \\ (x,j) &\longmapsto x, & (x,j) &\longmapsto x, & (x,j) &\longmapsto j, \end{aligned} \tag{2}$$

and we want to construct a subset of $\hat{L} = \{w \in h^{-1}(L) \mid \eta_0(\pi_{Y_0}(w)) = \eta_1(\pi_{Y_1}(w))\}$ in $\mathsf{Alg}(\mathsf{F}_i)$. Observe that we cannot hope to find $\hat{L}$ itself in $\mathsf{Alg}(\mathsf{F}_i)$ in general. Take, for example, the context-free language $E = \{a^n b^n \mid n \ge 0\}$ and $X_0 = \{a\}$, $X_1 = \{b\}$, $\gamma_0(a) = 1$, $\gamma_1(b) = 1$. Then the language $\hat{E}$ would not be context-free. However, the language $E' = \{wg(w)^R \mid w \in \{(a,0),(a,1)\}^*\}$, where g is the morphism with $(a,j) \mapsto (b,j)$ for $j = 0,1$, is context-free. Although it is only a proper subset of $\hat{E}$, it is large enough to satisfy $\pi_{Y_i}(E') = \pi_{Y_i}(\hat{E}) = \pi_{Y_i}(h^{-1}(E))$ for $i = 0,1$. We will see that in order to construct Parikh annotations, it suffices to use such under-approximations of $\hat{L}$.

Derivation trees and matchings In this work, by an *X-labeled tree*, we mean a finite ordered unranked tree in which each node carries a label from $X \cup \{\varepsilon\}$ for an alphabet X. For each node, there is a linear order on the set of its children. For each node x, we write $\mathsf{c}(x) \in X^*$ for the word obtained by reading the labels of x's children in this order. Furthermore, $\mathsf{yield}(x) \in X^*$ denotes the word obtained by reading leaf labels below the node x according to the linear order induced on the leaves. Moreover, if r is the root of t, we also write $\mathsf{yield}(t)$ for $\mathsf{yield}(r)$. The *height* of a tree is the maximal length of a path from the root to a leaf, i.e. a tree consisting of a single node has height 0. A *subtree* of a tree t is the tree consisting of all nodes below some node x of t. If x is a child of t's root, the subtree is a *direct subtree.*

Let $G = (N,T,P,S)$ be a $\mathcal{C}$-grammar. A *partial derivation tree (for G)* is an $(N \cup T)$-labeled tree t in which

(i) each inner node x has a label $A \in N$ and there is some $A \to L$ in P with $\mathsf{c}(x) \in L$, and
(ii) no ε-labeled node has a sibling.

If, in addition, the root is labeled S and every leaf is labeled by $T \cup \{\varepsilon\}$, it is called a *derivation tree for G*.

Let t be a tree whose leaves are $X \cup \{\varepsilon\}$-labeled. Let L_i denote the set of X_i-labeled leaves of t. An *arrow collection for t* is a finite set A together with maps $\nu_i\colon A \to L_i$ for $i = 0,1$. Hence, A can be thought of as a set of arrows pointing from X_0-labeled leaves to X_1-labeled leaves. We say an arrow $a \in A$ is *incident* to a leaf ℓ if $\nu_0(a) = \ell$ or $\nu_1(a) = \ell$. If ℓ is a leaf, then $d_A(\ell)$ denotes the number of arrows incident to ℓ. More generally, for a subtree s of t, $d_A(s)$ denotes the number of arrows incident to some leaf in s and some leaf outside of s. A is called a *k-matching* if

(i) each leaf labeled $x \in X_i$ has precisely $\gamma_i(x)$ incident arrows, and
(ii) $d_A(s) \le k$ for every subtree s of t.

The following lemma applies Lemma 11. The latter implies that for nodes x of a derivation tree, the balance $\gamma_0(\pi_{X_0}(\mathsf{yield}(x))) - \gamma_1(\pi_{X_1}(\mathsf{yield}(x)))$ is bounded. This can be used to construct k-matchings in a bottom-up manner.

▶ **Lemma 12.** *Let $X = X_0 \uplus X_1$ and $\gamma_i \colon X_i^* \to \mathbb{N}$ for $i = 0, 1$ be a morphism. Let G be a reduced F_i-grammar with $\mathsf{L}(G) \subseteq X^*$ and $\gamma_0(\pi_{X_0}(w)) = \gamma_1(\pi_{X_1}(w))$ for every $w \in \mathsf{L}(G)$. Then one can compute a bound k such that each derivation tree of G admits a k-matching.*

We are now ready to construct the approximations necessary for obtaining PAIM.

▶ **Proposition 13** (Consistent substitution). *Let $X = X_0 \uplus X_1$ and $\gamma_i \colon X_i^{\oplus} \to \mathbb{N}$ for $i = 0, 1$ be a morphism. Let $L \in \mathsf{Alg}(\mathsf{F}_i)$, $L \subseteq X^*$, be a language with $\gamma_0(\pi_{X_0}(w)) = \gamma_1(\pi_{X_1}(w))$ for every $w \in L$. Furthermore, let Y_i, h_i, η_i for $i = 0, 1$ and Y, h be defined as in Eq.* (1) *and Eq.* (2)*. Moreover, let L be given by a reduced grammar. Then one can construct a language $L' \in \mathsf{Alg}(\mathsf{F}_i)$, $L' \subseteq Y^*$, with*

(i) *$L' \subseteq h^{-1}(L)$,*
(ii) *$\pi_{Y_i}(L') = \pi_{Y_i}(h^{-1}(L))$ for $i = 0, 1$,*
(iii) *$\eta_0(\pi_{Y_0}(w)) = \eta_1(\pi_{Y_1}(w))$ for every $w \in L'$.*

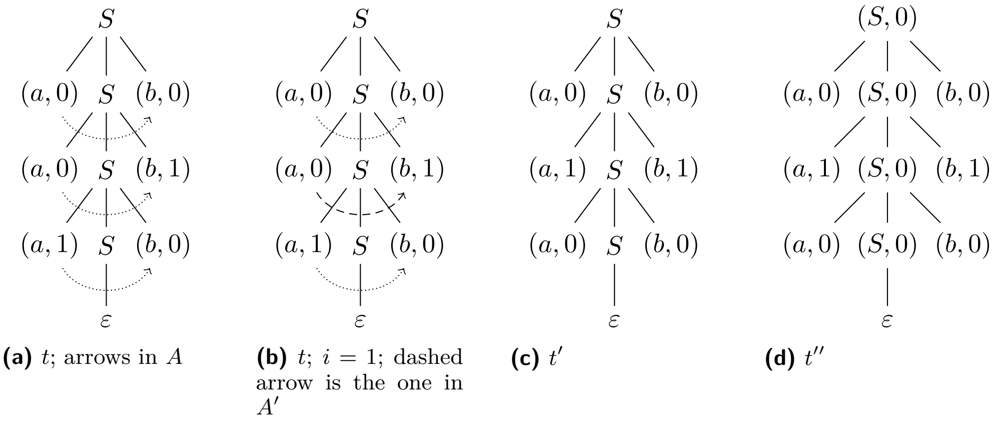
Proof. Let $G_0 = (N, X, P_0, S)$ be a reduced F_i-grammar with $\mathsf{L}(G_0) = L$. Let $G_1 = (N, Y, P_1, S)$ be the grammar with $P_1 = \{A \to \hat{h}^{-1}(K) \mid A \to K \in P_0\}$, where $\hat{h} \colon (N \cup Y)^* \to (N \cup X)^*$ is the extension of h that fixes N. With $L_1 = \mathsf{L}(G_1)$, we have $L_1 = h^{-1}(L)$.

According to Lemma 12, we can find a $k \in \mathbb{N}$ such that every derivation tree of G_0 admits a k-matching. With this, let $F = \{z \in \mathbb{Z} \mid |z| \leq k\}$, $N_2 = N \times F$, and η be the morphism $\eta \colon (N_2 \cup Y)^* \to \mathbb{Z}$ with $(A, z) \mapsto z$ for $(A, z) \in N_2$, and $y \mapsto \eta_0(\pi_{Y_0}(y)) - \eta_1(\pi_{Y_1}(y))$ for $y \in Y$. Moreover, let $g \colon (N_2 \cup Y)^* \to (N \cup Y)^*$ be the morphism with $g((A, z)) = A$ for $(A, z) \in N_2$ and $g(y) = y$ for $y \in Y$. This allows us to define the set of productions $P_2 = \{(A, z) \to g^{-1}(L) \cap \eta^{-1}(z) \mid A \to K \in P_1\}$. Note that since F_i is an effective Presburger closed full semi-trio, we have effectively $g^{-1}(K) \cap \eta^{-1}(z) \in \mathsf{F}_i$ for $K \in \mathsf{F}_i$. Finally, let G_2 be the grammar $G_2 = (N_2, Y, P_2, (S, 0))$. We claim that $L' = \mathsf{L}(G_2)$ has the desired properties. Since $L' \subseteq L_1 = h^{-1}(L)$, Item 1 is satisfied. Furthermore, the construction guarantees that for a production $(A, z) \to w$ in G_2, we have $\eta(w) = z$. In particular, every $w \in Y^*$ with $(S, 0) \Rightarrow_{G_2}^* w$ exhibits $\eta_0(\pi_{Y_0}(w)) - \eta_1(\pi_{Y_1}(w)) = \eta(w) = 0$. Thus, we have shown Item 3.

Note that the inclusion "$\subseteq$" of Item 2 follows from Item 1. In order to prove "$\supseteq$", we shall use k-matchings in G_0 to construct derivations in G_2. See Fig. 1 for an example of the following construction of derivation trees. Let $w \in h^{-1}(L) = \mathsf{L}(G_1)$ and consider a derivation tree t for w in G_1. Let $\bar{t}$ be the $(N \cup X)$-tree obtained from t by replacing each leaf label $y \in Y$ by $h(y)$. Then $\bar{t}$ is a derivation tree of G_0 and admits a k-matching $\bar{A}$. Since $\bar{t}$ and t are isomorphic up to labels, we can obtain a corresponding arrow collection A in t (see Fig. 1a).

Let L_i denote the set of Y_i-labeled leaves of t for $i = 0, 1$. Now fix $i \in \{0, 1\}$. We choose a subset $A' \subseteq A$ as follows. Since $\bar{A}$ is a k-matching, each leaf $\ell \in L_i$ of t has precisely $\gamma_i(h(\lambda(\ell))) \geq \eta_i(\lambda(\ell))$ incident arrows in A. For each such $\ell \in L_i$, we include some arbitrary choice of $\eta_i(\lambda(\ell))$ arrows in A' (see Fig. 1b). The tree t' is obtained from t by changing the label of each leaf $\ell \in L_{1-i}$ from (x, j) to (x, j'), where j' is the number of arrows in A' incident to ℓ (see Fig. 1c). Note that since we only change labels of leaves in L_{1-i}, we have $\pi_{Y_i}(\mathsf{yield}(t')) = \pi_{Y_i}(\mathsf{yield}(t)) = \pi_{Y_i}(w)$.

For every subtree s of t', we define $\beta(s) = \eta_0(\pi_{Y_0}(\mathsf{yield}(s))) - \eta_1(\pi_{Y_1}(\mathsf{yield}(s)))$. By construction of A', each leaf $\ell \in L_j$ has precisely $\eta_j(\lambda(\ell))$ incident arrows in A' for $j = 0, 1$.



(a) t; arrows in A

(b) t; $i=1$; dashed arrow is the one in A'

(c) t'

(d) t''

Figure 1 Derivation trees in the proof of Proposition 13 for the context-free grammar G with productions $S \to aSb$, $S \to \varepsilon$ and $X_0 = \{a\}$, $X_1 = \{b\}$, $\gamma_0(a) = \gamma_1(b) = 1$.

Therefore,

$$\beta(s) = \sum_{\ell \in L_0 \cap s} d_{A'}(\ell) - \sum_{\ell \in L_1 \cap s} d_{A'}(\ell). \tag{3}$$

The absolute value of the right hand side of this equation is at most $d_{A'}(s)$ and hence

$$|\eta_0(\pi_{Y_0}(\mathsf{yield}(s))) - \eta_1(\pi_{Y_1}(\mathsf{yield}(s)))| = |\beta(s)| \leq d_{A'}(s) \leq d_A(s) \leq k \tag{4}$$

since $\bar{A}$ is a k-matching. In the case $s = t'$, Eq. (3) also tells us that

$$\eta_0(\pi_{Y_0}(\mathsf{yield}(t'))) - \eta_1(\pi_{Y_1}(\mathsf{yield}(t'))) = \sum_{\ell \in L_0} d_{A'}(\ell) - \sum_{\ell \in L_1} d_{A'}(\ell) = 0. \tag{5}$$

Let t'' be the tree obtained from t' as follows: For each N-labeled node x of t', we replace the label B of x with $(B, \beta(s))$, where s is the subtree below x (see Fig. 1d). By Eq. (4), this is a symbol in N_2. The root node of t'' has label $(S,0)$ by Eq. (5). Furthermore, it follows by an induction on the height of subtrees that if (B,z) is the label of a node x, then $z = \eta(\mathsf{c}(x))$. Hence, the tree t'' is a derivation tree of G_2. This means $\pi_{Y_i}(w) = \pi_{Y_i}(\mathsf{yield}(t')) = \pi_{Y_i}(\mathsf{yield}(t'')) \in L(G_2) = L'$, completing the proof of Item 2. ◀

Proposition 13 now allows us to construct PAIM for languages $\sigma(L)$, where σ is a letter substitution. The essential idea is to use a PAIM $(K, C, P, (P_c)_{c \in C}, \varphi, \diamond)$ for L and then apply Proposition 13 to K with $X_0 = Z \cup \{\diamond\}$ and $X_1 = C \cup P$. One can clearly assume that a single letter a from Z is replaced by $\{a, b\} \subseteq Z'$. We can therefore choose $\gamma_0(w)$ to be the number of a's in w and $\gamma_1(w)$ to be the number of a's represented by symbols from $C \cup P$ in w. Then the counting property of K entails $\gamma_0(w) = \gamma_1(w)$ for $w \in K$ and thus applicability of Proposition 13. Item 2 then yields the projection property for $i = 0$ and the commutative projection property for $i = 1$ and Item 3 yields the counting property for the new PAIM.

▶ **Lemma 14** (Letter substitution). *Let $\sigma\colon Z \to \mathcal{P}(Z')$ be a letter substitution. Given $i \in \mathbb{N}$ and a PAIM for $L \in \mathsf{G}_i$ in G_i, one can construct a PAIM in G_i for $\sigma(L)$.*

The basic idea for the case of general substitutions is to replace each x by a PAIM for $\sigma(x)$. Here, Lemma 14 allows us to assume that the PAIM for each $\sigma(x)$ is linear. However, we have to make sure that the number of occurrences of $\diamond$ remains bounded.

▶ **Lemma 15** (Substitutions). *Let $L \subseteq X^*$ in G_i and σ be a G_i-substitution. Given a PAIM in G_i for L and for each $\sigma(x)$, $x \in X$, one can construct a PAIM for $\sigma(L)$ in G_i.*

The next step is to construct PAIM for languages $\mathsf{L}(G)$, where G has just one nonterminal S and PAIM are given for the right-hand-sides. Here, it suffices to obtain a PAIM for $\mathsf{SF}(G)$ in the case that S occurs in every word on the right hand side: Then $\mathsf{L}(G)$ can be obtained from $\mathsf{SF}(G)$ using a substitution. Applying $S \to R$ then means that for some $w \in R$, $\Psi(w) - S$ is added to the Parikh image of the sentential form. Therefore, computing a PAIM for $\mathsf{SF}(G)$ is akin to computing a semilinear representation for $S^{\oplus}$, where S is semilinear.

▶ **Lemma 16** (One nonterminal). *Let G be a G_i-grammar with one nonterminal. Furthermore, suppose PAIM in G_i are given for the right-hand-sides in G. Then we can construct a PAIM for $\mathsf{L}(G)$ in G_i.*

Using Lemmas 15 and 16, we can now construct PAIM recursively with respect to the number of nonterminals in G.

▶ **Lemma 17** (PAIM for algebraic extensions). *Given $i \in \mathbb{N}$ and an F_i-grammar G, along with a PAIM in F_i for each right hand side, one can construct a PAIM for $\mathsf{L}(G)$ in G_i.*

The last step is to compute PAIM for languages in $\mathsf{SLI}(\mathsf{G}_i)$. Then, Theorem 10 follows.

▶ **Lemma 18** (PAIM for semilinear intersections). *Given $i \in \mathbb{N}$, a language $L \subseteq X^*$ in G_i, a semilinear set $S \subseteq X^{\oplus}$, and a morphism $h\colon X^* \to Y^*$, along with a PAIM in G_i for L, one can construct a PAIM for $h(L \cap \Psi^{-1}(S))$ in $\mathsf{SLI}(\mathsf{G}_i)$.*

5 Computing downward closures

The procedure for computing downward closures works recursively with respect to the hierarchy $\mathsf{F}_0 \subseteq \mathsf{G}_0 \subseteq \cdots$. For languages in $\mathsf{G}_i = \mathsf{Alg}(\mathsf{F}_i)$, we use an idea by van Leeuwen [17], who proved that downward closures are computable for $\mathsf{Alg}(\mathcal{C})$ if and only if this is the case for $\mathcal{C}$. This means we can compute downward closures for G_i if we can compute them for F_i. For the latter, we use Lemma 19, which is based on the following idea. Using a PAIM for L in G_i, one constructs a language $L' \supseteq L \cap \Psi^{-1}(S)$ in which every word admits insertions that yield a word in $L \cap \Psi^{-1}(S)$, meaning that $L'{\downarrow} = (L \cap \Psi^{-1}(S)){\downarrow}$. Here, L' is obtained from the PAIM using a rational transduction, which implies $L' \in \mathsf{G}_i$.

▶ **Lemma 19.** *Given $i \in \mathbb{N}$, a language $L \subseteq X^*$ in G_i, and a semilinear set $S \subseteq X^{\oplus}$, one can compute a language $L' \in \mathsf{G}_i$ with $L'{\downarrow} = (L \cap \Psi^{-1}(S)){\downarrow}$.*

Proof. We call $\alpha \in X^{\oplus}$ a *submultiset* of $\beta \in X^{\oplus}$ if $\alpha(x) \leq \beta(x)$ for each $x \in X$. In analogy with words, we write $T{\downarrow}$ for the set of all submultisets of elements of T for $T \subseteq X^{\oplus}$. We use Theorem 10 to construct a PAIM $(K, C, P, (P_c)_{c \in C}, \varphi, \diamond)$ for L in G_i. For each $c \in C$, consider the set $S_c = \{\mu \in P_c^{\oplus} \mid \varphi(c + \mu) \in S\}$. Since $\leq$ is a well-quasi-ordering on $X^{\oplus}$ [6], membership in $S_c{\downarrow}$ can be characterized by a finite set of forbidden submultisets, which is Presburger definable and thus computable. Therefore, the language $\Psi^{-1}(S_c{\downarrow})$ is effectively regular. Hence, the language

$$L' = \{\pi_X(cv) \mid c \in C,\ cv \in K,\ \pi_{P_c}(v) \in \Psi^{-1}(S_c{\downarrow})\}.$$

effectively belongs to G_i, since G_i is an effective full semi-AFL. We claim that $L \cap \Psi^{-1}(S) \subseteq L' \subseteq (L \cap \Psi^{-1}(S))\downarrow$. The latter clearly implies $L'\downarrow = (L \cap \Psi^{-1}(S))\downarrow$.

The counting property of the PAIM entails the inclusion $L \cap \Psi^{-1}(S) \subseteq L'$. In order to show $L' \subseteq (L \cap \Psi^{-1}(S))\downarrow$, suppose $w \in L'$. Then there is a $cv \in K$ with $w = \pi_X(cv)$ and $\pi_{P_c}(v) \in \Psi^{-1}(S_c\downarrow)$. This means there is a $\nu \in P_c^{\oplus}$ with $\Psi(\pi_{P_c}(v)) + \nu \in S_c$. The insertion property of $(K, C, P, (P_c)_{c \in C}, \varphi, \diamond)$ allows us to find a word $v' \in L$ such that

$$\Psi(v') = \Psi(\pi_X(cv)) + \varphi(\nu), \qquad \pi_{X \cup \{\diamond\}}(cv) \preceq_\diamond v'. \tag{6}$$

By definition of S_c, the first part of Eq. (6) implies that $\Psi(v') \in S$. The second part of Eq. (6) means in particular that $w = \pi_X(cv) \preceq v'$. Thus, we have $w \preceq v' \in L \cap \Psi^{-1}(S)$. ◀

▶ **Theorem 20.** *Given a language L in F, one can compute a finite automaton for $L\downarrow$.*

Proof. We perform the computation recursively with respect to the level of the hierarchy.

- If $L \in \mathsf{F}_0$, then L is finite and we can clearly compute $L\downarrow$.
- If $L \in \mathsf{F}_i$ with $i \geq 1$, then $L = h(L' \cap \Psi^{-1}(S))$ for some $L' \subseteq X^*$ in G_{i-1}, a semilinear set $S \subseteq X^{\oplus}$, and a morphism h. Since $h(M)\downarrow = h(M\downarrow)\downarrow$ for any $M \subseteq X^*$, it suffices to describe how to compute $(L' \cap \Psi^{-1}(S))\downarrow$. Using Lemma 19, we construct a language $L'' \in \mathsf{G}_{i-1}$ with $L''\downarrow = (L' \cap \Psi^{-1}(S))\downarrow$ and then recursively compute $L''\downarrow$.
- If $L \in \mathsf{G}_i$, then L is given by an F_i-grammar G. Using recursion, we compute the downward closure of each right-hand-side of G. We obtain a new REG-grammar G' by replacing each right-hand-side in G with its downward closure. Then $\mathsf{L}(G')\downarrow = L\downarrow$. Since we can construct a context-free grammar for $\mathsf{L}(G')$, we can compute $\mathsf{L}(G')\downarrow$ using the available algorithms by van Leeuwen [16] or Courcelle [5].

◀

6 Strictness of the hierarchy

In this section, we present another application of Parikh annotations. Using PAIM, one can show that the inclusions $\mathsf{F}_0 \subseteq \mathsf{G}_0 \subseteq \mathsf{F}_1 \subseteq \mathsf{G}_1 \subseteq \cdots$ in the hierarchy are, in fact, all strict. It is of course easy to see that $\mathsf{F}_0 \subsetneq \mathsf{G}_0 \subsetneq \mathsf{F}_1$, since F_0 contains only finite sets and F_1 contains, for example, $\{a^n b^n c^n \mid n \geq 0\}$. In order to prove strictness at higher levels, we present two transformations: The first turns a language from $\mathsf{F}_i \setminus \mathsf{G}_{i-1}$ into one in $\mathsf{G}_i \setminus \mathsf{F}_i$ (Proposition 21) and the second turns one from $\mathsf{G}_i \setminus \mathsf{F}_i$ into one in $\mathsf{F}_{i+1} \setminus \mathsf{G}_i$ (Proposition 25).

The essential idea of the next proposition is as follows. For the sake of simplicity, assume $(L\#)^* = L' \cap \Psi^{-1}(S)$ for $L' \in \mathcal{C}$, $L' \subseteq (X \cup \{\#\})^*$. Consider a PAIM $(K', C, P, (P_c)_{c \in C}, \varphi, \diamond)$ for L' in $\mathcal{C}$. Similar to Lemma 19, we obtain from K' a language $\hat{L} \subseteq (X \cup \{\#, \diamond\})^*$ in $\mathcal{C}$ such that every member of $\hat{L}$ admits an insertion at $\diamond$ that yields a word from $(L\#)^* = L' \cap \Psi^{-1}(S)$. Using a rational transduction, we can then pick all words that appear between two # in some member of $\hat{L}$ and contain no $\diamond$. Since there is a bound on the number of $\diamond$ in K' (and hence in $\hat{L}$), every word from L has to occur in this way. On the other hand, since inserting at $\diamond$ yields a word in $(L\#)^*$, every such word without $\diamond$ must be in L.

▶ **Proposition 21.** *Let $\mathcal{C}$ be a full trio such that every language in $\mathcal{C}$ has a PAIM in $\mathcal{C}$. Moreover, let X be an alphabet with $\# \notin X$. If $(L\#)^* \in \mathsf{SLI}(\mathcal{C})$ for $L \subseteq X^*$, then $L \in \mathcal{C}$.*

Using induction on the structure of a rational expression, it is not hard to show that we can construct PAIM for regular languages. This means Propositions 2 and 21 imply the following, which might be of independent interest.

▶ **Corollary 22.** *Let $L \subseteq X^*$, $\# \notin X$, and $(L\#)^* \in \mathsf{VA}(\mathbb{Z}^n)$. Then L is regular.*

In order to prove Proposition 25, we need a new concept. A bursting grammar is one in which essentially (meaning: aside from a subsequent replacement by terminal words of bounded length) the whole word is generated in a single application of a production.

▶ **Definition 23.** *Let $\mathcal{C}$ be a language class and $k \in \mathbb{N}$. A $\mathcal{C}$-grammar G is called k-*bursting *if for every derivation tree t for G and every node x of t we have: $|\mathsf{yield}(x)| > k$ implies $\mathsf{yield}(x) = \mathsf{yield}(t)$. A grammar is said to be* bursting *if it is k-bursting for some $k \in \mathbb{N}$.*

▶ **Lemma 24.** *If $\mathcal{C}$ is a union closed full semi-trio and G a bursting $\mathcal{C}$-grammar, then $\mathsf{L}(G) \in \mathcal{C}$.*

The essential idea for Proposition 25 is the following. We construct a $\mathcal{C}$-grammar G' for L by removing from a $\mathcal{C}$-grammar G for $M = (L \sqcup \{a^n b^n c^n \mid n \geq 0\}) \cap a^*(bX)^*c^*$ all terminals a, b, c. Using Lemma 11, one can then show that G' is bursting.

▶ **Proposition 25.** *Let $\mathcal{C}$ be a union closed full semi-trio and let $a, b, c \notin X$ and $L \subseteq X^*$. If $L \sqcup \{a^n b^n c^n \mid n \geq 0\} \in \mathsf{Alg}(\mathcal{C})$, then $L \in \mathcal{C}$.*

▶ **Theorem 26.** *For $i \in \mathbb{N}$, define the alphabets $X_0 = \emptyset$, $Y_i = X_i \cup \{\#_i\}$, $X_{i+1} = Y_i \cup \{a_{i+1}, b_{i+1}, c_{i+1}\}$. Moreover, define $U_i \subseteq X_i^*$ and $V_i \subseteq Y_i^*$ as $U_0 = \{\varepsilon\}$, $V_i = (U_i\#_i)^*$, and $U_{i+1} = V_i \sqcup \{a_{i+1}^n b_{i+1}^n c_{i+1}^n \mid n \geq 0\}$ for $i \geq 0$. Then $V_i \in \mathsf{G}_i \setminus \mathsf{F}_i$ and $U_{i+1} \in \mathsf{F}_{i+1} \setminus \mathsf{G}_i$.*

References

1 Parosh Aziz Abdulla, Luc Boasson, and Ahmed Bouajjani. Effective lossy queue languages. In *Proc. of ICALP 2001*, volume 2076 of *LNCS*, pages 639–651. Springer, 2001.

2 Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer. Context-bounded analysis for concurrent programs with dynamic creation of threads. In *Proc. of TACAS 2009*, volume 5505 of *LNCS*, pages 107–123. Springer, 2009.

3 Georg Bachmeier, Michael Luttenberger, and Maximilian Schlund. Finite automata for the sub- and superword closure of cfls: Descriptional and computational complexity, 2015. To appear in: Proceedings of LATA 2015.

4 P. Buckheister and Georg Zetzsche. Semilinearity and context-freeness of languages accepted by valence automata. In *Proc. of MFCS 2013*, volume 8087 of *LNCS*, pages 231–242. Springer, 2013.

5 Bruno Courcelle. On constructing obstruction sets of words. *Bulletin of the EATCS*, 44:178–186, 1991.

6 Leonard Eugene Dickson. Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. *American Journal of Mathematics*, 35(4):413–422, 1913.

7 Hermann Gruber, Markus Holzer, and Martin Kutrib. The size of Higman-Haines sets. *Theoretical Computer Science*, 387(2):167–176, 2007.

8 Hermann Gruber, Markus Holzer, and Martin Kutrib. More on the size of higman-haines sets: effective constructions. *Fundamenta Informaticae*, 91(1):105–121, 2009.

9 Peter Habermehl, Roland Meyer, and Harro Wimmel. The downward-closure of Petri net languages. In *Proc. of ICALP 2010*, volume 6199 of *LNCS*, pages 466–477. Springer, 2010.

10 Graham Higman. Ordering by divisibility in abstract algebras. *Proceedings of the London Mathematical Society. Third Series*, 2:326–336, 1952.

11 Prateek Karandikar and Philippe Schnoebelen. On the state complexity of closures and interiors of regular languages with subwords. In *Proc. of DCFS 2014*, volume 8614 of *LNCS*, pages 234–245. Springer, 2014.

12 Eryk Kopczynski and Anthony Widjaja To. Parikh images of grammars: Complexity and applications. In *Proc. of LICS 2010*, pages 80–89. IEEE, 2010.

13 Zhenyue Long, Georgel Calin, Rupak Majumdar, and Roland Meyer. Language-theoretic abstraction refinement. In *Proc. of FASE 2012*, volume 7212 of *LNCS*, pages 362–376. Springer, 2012.

14 Richard Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297(1-3):337–354, 2003.

15 Alexander Okhotin. On the state complexity of scattered substrings and superstrings. *Fundamenta Informaticae*, 99(3):325–338, 2010.

16 Jan van Leeuwen. A generalisation of Parikh's theorem in formal language theory. In *Proc. of ICALP 1974*, volume 14 of *LNCS*, pages 17–26. Springer, 1974.

17 Jan van Leeuwen. Effective constructions in well-partially-ordered free monoids. *Discrete Mathematics*, 21(3):237–252, 1978.

18 Georg Zetzsche. Computing downward closures for stacked counter automata.

19 Georg Zetzsche. Silent transitions in automata with storage. In *Proc. of ICALP 2013*, volume 7966 of *LNCS*, pages 434–445. Springer, 2013.