Electronic Colloquium on Computational Complexity, Report No. 18 (2003)

Functions Computable in Polynomial Space

Matthias Galota and Heribert Vollmer

Theoretische Informatik
Universität Hannover
Appelstraße 4
30167 Hannover, Germany

(galota|vollmer)@informatik.uni-hannover.de

Abstract

We show that the class of integer-valued functions computable by polynomial-space Turing machines is exactly the class of functions f for which there is a nondeterministic polynomial-*time* Turing machine that on input x outputs a 3×3 matrix with entries from $\{-1, 0, 1\}$ on each of its paths such that $f(x)$ is exactly the upper left entry in the product of all these matrices in an order of the paths to be made precise below. Along the way we obtain characterizations of FPSPACE in terms of arithmetic circuits and straight-line programs.

Keywords: polynomial space, complexity class of functions, bottleneck machines, leaf languages, arithmetic circuits, straight-line programs

1 Introduction

The class PSPACE of languages accepted by deterministic polynomial-space Turing machines has a remarkable characterization in terms of *bottleneck machines* [CF91]: Take an arbitrary regular language B whose syntactic monoid is not solvable, for example let B consist of those sequences of permutations on 5 elements that multiply out to the identity. Then, for every language $L \in$ PSPACE there is a nondeterministic polynomial-time Turing machine that, given any input word x, produces as output a symbol from a fixed alphabet on each of its paths. Take the word obtained by concatenating these symbols from left to right (in some fixed order of paths, e.g., the one induced by the order of nondeterministic choices in the machines transition table) and call it the *leaf string* of M on input x. Then, $x \in L$ iff the leaf string of M on x belongs to B. This characterization—essentially nothing else than a translation version of Barrington's seminal characterization of NC^1 by bounded-width branching programs [Bar89] into the context of alternating polynomial-time Turing machines—is referred to as "bottleneck" characterization, since we can cut up a PSPACE computation into polynomial-time slices (namely, the computations of the single paths of an NPTM) such that each of these subcomputations passes only a constant amount of information through a bottleneck to its right neighbor (the state of a finite automaton, or an element from S_5). Because the leaf string of an NTM is central in this approach, it is also referred to as the *leaf language characterization* of PSPACE [HLS$^+$93], see also [Pap94, Vol99b].

This paper arose from the search for a similar characterization of the class FPSPACE of all *functions* computable in polynomial space. We were looking for a simple function $\mathfrak{F}$, such that for every $h \in$ FPSPACE there is a polynomial-time nondeterministic Turing machine M with the property that for every input word x, the value $h(x)$ is the value of $\mathfrak{F}$ applied to the leaf string of

ISSN 1433-8092

M on x.

Being a little bit more general, given any function F that evaluates leaf strings (we call F a *leaf function*), define the class F-FP as the class of all functions h for which there is a NPTM M such that $h(x)$ equals the value of F applied to the leaf string of $M(x)$ (a more formal definition is given in Sect. 2). The question now is if there is a "simple" leaf function $\mathfrak{F}$ such that FPSPACE $= \mathfrak{F}$-FP. $\mathfrak{F}$ should be "multiplication like" in the sense that each path contributes with some finite information that is "multiplied" to the result obtained from the path to the left, and the "product" obtained as a result of this multiplication will then be propagated to the path to the right.

From the leaf language characterization of the (language) class PSPACE, it is actually not too hard to conclude that there is a finite automaton that, given a leaf string as input, can compute the output of the corresponding FPSPACE-function:

Theorem 1.1. *There is a finite automaton M with output such that for the function f_M computed by M, we have: f_M-FP = FPSPACE.*

A proof for this Theorem is given in section 4.

However, we were not satisfied with the just given result. It is very much of a formal language theoretic nature, and does not address the nice algebraic properties that the class FPSPACE shares. A paper by Richard Ladner [Lad89] shows that the class FPSPACE coincides with the counting class #APTIME. Say that a *proof tree* of an alternating Turing machine M is a minimal edge-induced subtree of the computation tree of M that proves that M accepts its input, in the same vein as an accepting path of a nondeterministic machine proves that the machine accepts its input (a formal definition is given in Sect. 2). Now, a function f belongs to #APTIME if there is an alternating polynomial-time machine M such that, for every x, $f(x)$ equals the number of proof trees of M working on input x.

Counting proof trees of alternating machines corresponds in a nice way to evaluating arithmetic circuits over the natural numbers; hence Ladner's result indirectly yields a nice algebraic characterization of PSPACE. It was our goal to obtain a leaf function characterization of FPSPACE that uses the spirit of Ladner's result. We obtained the following:

Theorem 1.2. *Let $\mathfrak{F}$ be the function that, given a sequence of 3×3 matrices with entries from $\{-1, 0, 1\}$, multiplies out this sequence and outputs as result the upper left entry in the product. Then, $\mathfrak{F}$-FP = FPSPACE.*

The proof of this result relies on a stronger characterization of FPSPACE in terms of counting functions than the one given in Ladner's paper [Lad89]. We first move to the slightly more general setting of polynomial space Turing machines that compute integer valued functions, i.e., we allow negative values, and show that this class coincides with the class of functions that can be obtained as differences of two functions counting proof trees of alternating machines. Then, we combine relations among alternating Turing machines, arithmetic circuits over the integers, and straight-line programs. Our main result then follows by expressing the computation of a straight-line program in terms of multiplication of small-dimensional matrices. Along the way, we thus obtain characterizations of FPSPACE using all these different computation models; e.g., we show that FPSPACE-functions are exactly those that can be computed by exponential size straight-line programs that use only 3 registers.

Galperin and Wigderson [GW83] investigated problems whose instances are not given in the usual string representation but encoded by Boolean circuits. The *succinct version* of a language L is defined to be the class of all circuits that encode strings in L. A consequence of our result concerns complete problems for PSPACE: It follows that multiplication of constant dimension matrices over $\{-1, 0, 1\}$ is complete for PSPACE if the matrices are presented in a succinct way.

In the next section, we introduce the different computation models we make use of in this paper: counting Turing machines, leaf functions, arithmetic circuits, straight-line programs, and matrix programs. The latter three are non-uniform models—hence we have to look for an appropriate notion of uniformity. Our main result holds for a very strict uniformity condition which we call *full binary tree uniformity.* Section 3 then contains the proof of Theorem 1.2 making use of a number of simulations among the different models defined previously.

2 Preliminaries

2.1 Complexity Classes of Functions

We fix the alphabet $\Sigma = \{0, 1\}$. Functions computed by deterministic Turing machines are functions $f\colon \Sigma^* \to \Sigma^*$. Making use of the well-known bijections between Σ^* and the sets $\mathbb{N}$ of natural numbers and $\mathbb{Z}$ of integers, we will, depending on the context, consider Turing machines computing functions $f\colon \Sigma^* \to \mathbb{N}$ or $f\colon \Sigma^* \to \mathbb{Z}$. We call the class of all functions $f\colon \Sigma^* \to \mathbb{Z}$ which can be computed by polynomial space TMs FPSPACE. Here, the number of cells used on the output tape does not contribute to the space requirement of the machine; hence the output of a FPSPACE function may be exponentially long compared to the input length. The class of functions $f\colon \Sigma^* \to \mathbb{N}$ computable by PSPACE TMs will be referred to as FPSPACE_+.

We will make use of some counting function classes: For a TM M and input x, define $acc(M, x)$ to be the number of distinct accepting and $rej(M, x)$ to be the number of distinct rejecting computation paths of M on x. Let M be an NPSPACE TM. Define #PSPACE to be the class of functions $f\colon \Sigma^* \to \mathbb{N}$ for which there is an NPSPACE TM M where $f(x) = acc(M, x)$ for all x. Define Gap-PSPACE to be the class of functions $f\colon \Sigma^* \to \mathbb{Z}$ for which there is an NPSPACE TM M where $f(x) = acc(M, x) - rej(M, x)$ for all x.

Now let M be an APTIME TM. An *accepting computation tree* (or, *proof tree*) [VT89] on input x is a subtree T' of the computation tree $T_{M(x)}$ of M on x whose root is the root of $T_{M(x)}$, whose leaves are accepting leaves of $T_{M(x)}$ and whose nodes can all be obtained respecting the following conditions:

- Every existential node in T' has exactly one successor node in T'.
- Every universal node in T' has all its successors in $T_{M(x)}$ also in T'.

Define $\#(M, x)$ as the number of different accepting computation trees of M on input x and define #APTIME to be the class of functions $f\colon \Sigma^* \to \mathbb{N}$ such that there is an APTIME TM M where $f(x) = \#(M, x)$ for all x.

Proposition 2.1 [Lad89]. $\text{FPSPACE}_+ = \#\text{PSPACE} = \#\text{APTIME}$.

In [KSV00] a general framework to describe function classes was introduced, which closely resembles the leaf language framework, originally introduced to describe complexity classes of sets [BCS92, Ver93]. We will use a slightly simplified version, that suffices for our purpose here:

Let Γ be a finite alphabet and let Ω be the set of all finite vectors (sequences) of elements from Γ. A *leaf function* is a function $F\colon \Omega \to \mathbb{Z}$ that *evaluates* those sequences. (Leaf functions in the slightly more general setting of [KSV00] were called *generators* in that paper.) Every leaf function F defines the class F-FP as the class of functions $f\colon \Sigma^* \to \mathbb{Z}$ for which there exist polynomial-time computable functions $g\colon \Sigma^* \times \mathbb{N} \to \Gamma$ and $h\colon \Sigma^* \to \mathbb{N}$ such that for all $x \in \mathbb{N}$, the sequence $S_x = (g(x, 0), g(x, 1), \ldots, g(x, h(x)))$ belongs to Ω and $f(x) = F(S_x)$. The reader should think of $g(x, i)$ as the output of some nondeterministic polynomial-time Turing machine with input x on path i and of S_x as the sequence of outputs of that NPTM (i.e., the leaf string of $M(x)$) that is evaluated by the leaf function F.

Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be two function classes. We define $\mathcal{F}_1 - \mathcal{F}_2$ to be the class of differences of functions from $\mathcal{F}_1$ and $\mathcal{F}_2$ (not to be confused with $\mathcal{F}_1 \setminus \mathcal{F}_2$), i.e., $\{\, f \mid \text{there exist } f_1 \in \mathcal{F}_1 \text{ and } f_2 \in \mathcal{F}_2 \text{ with } f(x) = f_1(x) - f_2(x) \text{ for all } x \,\}$.

2.2 Non-uniform Computation Models

2.2.1 Arithmetic Circuits

Arithmetic circuits over the integers have input gates with values from $\{0, 1\}$ and constant gates 0, 1, and -1; their inner nodes are gates which compute addition and multiplication. An arithmetic circuit with n input gates computes a function $f\colon \{0,1\}^n \to \mathbb{Z}$ in the natural way.

Let $\overline{C} = (C_n)_{n\in\mathbb{N}}$ be a family of arithmetic circuits where C_n has n input gates and let $s, d\colon \mathbb{N} \to \mathbb{N}$ be functions. We say $\overline{C}$ is of *size* s (*depth* d) if, for every $n \in \mathbb{N}$, it holds that the number of gates of C_n is not larger than $s(n)$ (the length of the longest directed path in C_n is not larger than $d(n)$). Let $F\colon \{0,1\}^* \to \mathbb{Z}$ and $s, d\colon \mathbb{N} \to \mathbb{N}$ be functions. We say $F \in$ AC-SIZE-DEPTH(s, d) if there exists a family $\overline{C}$ of arithmetic circuits of size s and depth d, such that $f(x) = SP_{|x|}(x)$, for all $x \in \{0,1\}^*$.

2.2.2 Straight Line Programs

An n-input *straight-line program* using m registers is, following [BOC92] (Ben-Or and Cleve actually referred to this particular syntax of programs as *linear bijection straight-line programs*), a sequence of instructions $SP = (s_t)_{1\le t\le l}$, where each instruction s_t is of one of the following forms:

- $R_j \leftarrow R_j + c * R_i$,
- $R_j \leftarrow R_j - c * R_i$,
- $R_j \leftarrow R_j + x_k * R_i$,
- $R_j \leftarrow R_j - x_k * R_i$,

where $i, j \in \{1, \ldots, m\}$, $i \neq j$, $c \in \{0, 1\}$, $k \in \{1, \ldots, n\}$. The size of SP is defined to be l.

Such a program computes a function $f_{SP}\colon \{0,1\}^n \to \mathbb{Z}$ as follows: Let $x = x_1 \cdots x_n$. At the beginning of the computation, register R_1 has as contents the value 1 and all other registers have contents 0. Then the instructions $s_1; s_2; \ldots; s_l$ are performed in this order, changing the contents of the registers in the natural way. Finally, the contents of register R_1 is the result of the computation, i.e., the value of $f_{SP}(x)$.

Let $\overline{SP} = (SP_n)_{n\in\mathbb{N}}$ be a family of straight-line programs and $s\colon \mathbb{N} \to \mathbb{N}$ a function. We say $\overline{SP}$ is of *size* s if, for every $n \in \mathbb{N}$, it holds that the length of SP_n is not larger than $s(n)$. Let $F\colon \{0,1\}^* \to \mathbb{Z}$ and $s, r\colon \mathbb{N} \to \mathbb{N}$ be functions. We say $F \in$ SLP-SIZE-REG(s, r) if there exists a family $\overline{SP}$ of straight-line programs of size s, such that SP_n does not use more than $r(n)$ registers, for which holds $f(x) = SP_{|x|}(x)$, for all $x \in \mathbb{N}$.

2.2.3 Matrix Programs

A d-dimensional n-input matrix program is a sequence of $d \times d$ matrices $MP = (N_t)_{1\le t\le l}$, where the entries of each matrix consist of elements of the set $\{-1, 0, 1\} \cup \{x_1, \ldots, x_n, -x_1, \ldots, -x_n\}$ [CMTV98]. The length of the program is defined to be l.

Such a program computes a function $f_{MP}\colon \{0,1\}^n \to \mathbb{Z}$ as follows: Let $x = x_1 \cdots x_n$. The result of the computation of MP is

$$f_{MP}(x) = (1\ 0\ \ldots\ 0) \cdot \left(\prod_{t=1}^{l} N_t\right) \cdot \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix},$$

in other words: $f_{MP}(x)$ is the upper left entry of the matrix $\prod_{t=1}^{l} N_t$.

Let $\overline{MP} = (MP_n)_{n\in\mathbb{N}}$ be a family of matrix programs and $s\colon \mathbb{N} \to \mathbb{N}$ a function. We say $\overline{MP}$ is of *size* s if, for every $n \in \mathbb{N}$, it holds that the length of MP_n is not larger than $s(n)$. Let $F\colon \{0,1\}^* \to \mathbb{Z}$ and $s, d\colon \mathbb{N} \to \mathbb{N}$ be functions. We say $F \in$ MP-SIZE-DIM(s,d) if there exists a family $\overline{MP}$ of matrix programs of size s, such that MP_n is of dimension $d(n)$, for which holds $f(x) = MP_{|x|}(x)$, for all $x \in \mathbb{N}$.

2.2.4 Uniformity

One can assign to each computation tree $T_{M(x)}$ of an alternating TM M on input x an arithmetic circuit $C_{M(x)}$ by doing the following:

1) Each existential node of $T_{M(x)}$ is an addition gate in $C_{M(x)}$.
2) Each universal node of $T_{M(x)}$ is a multiplication gate in $C_{M(x)}$.
3) Two nodes which are connected by an edge in $T_{M(x)}$ are connected by a wire in $C_{M(x)}$.
4) If a path of $T_{M(x)}$ is accepting—which means that a 1 is printed on the leaf—the corresponding gate of $C_{M(x)}$ is a constant gate 1. If a path is rejecting, the corresponding gate is a constant gate 0.
5) The root of $T_{M(x)}$ is the output gate of $C_{M(x)}$.

It is easy to see that $\#(M,x) = C_{M(x)}$. (Note that the circuit does not have any input gates.)

We say M is *circuit preserving* if M produces full binary computation trees $T_{M(x)}$, for which holds that for all inputs x of equal length the $T_{M(x)}$ are equal except for the leaves, which may have different outputs.

If we want to describe arithmetic circuits with input gates we have to take a look at alternating transducers. An alternating transducer M on input x describes an arithmetic circuit $C_{M(x)}$ if it produces only outputs in $\{-1,0,1,2,3,4,\ldots,|x|+1\}$ on its leaves. Then the circuit described is as with TMs but with the following difference:

4) If, on a leaf of $T_{M(x)}$, the output is a number $2 \le i \le |x|+1$, then the corresponding gate of C_x will be an input gate labeled x_{i-1}, if it is a number $-1 \le i \le 1$, the corresponding gate will be a constant gate i.

The value computed by the circuit on input x is $C_{M(x)}(x)$.

If the transducer M produces identical computation trees for all inputs of the same length (which is *more* than circuit preserving), then we can call the produced arithmetic circuits $C_{M(|x|)}$. The transducer then describes a circuit family $\overline{C_M}$.

Now we say that a family $\overline{C}$ of arithmetic circuits is $\mathrm{U_{FBT}}$-uniform (full binary tree uniform) if it can be described in the above way by a polynomial-time alternating transducer, which produces only full binary computation trees.

A family $\overline{SLP}$ of straight-line programs is $\mathrm{U_{FBT}}$-uniform if there exists a polynomial-time nondeterministic transducer which produces only full binary computation trees and, for input x, outputs as a leaf string the straight-line program $SLP_{|x|}$ with the instructions as leaves.

The definition for matrix programs is analogous: We have to find a polynomial-time transducer which outputs the matrices of $MP_{|x|}$ on its leaves.

3 Result

We will make use of two technical lemmas:

Lemma 3.1. $\mathrm{P} \subseteq \mathrm{U_{FBT}\text{-}AC\text{-}SIZE\text{-}DEPTH}(2^{n^{O(1)}}, n^{O(1)})$.

Proof. We know that $\mathrm{P} = \mathrm{U_L\text{-}SIZE}(n^{O(1)})$ (see [Vol99a] Corollary 2.30.). Let $L \in \mathrm{P}$. Then there exists a transducer M which produces, on input 1^n, binary Boolean circuits B_n of size and height $p(n)$, for some polynomial p, which accept all x with $|x| = n$ and $x \in L$. Since the space complexity of M is $\log(p(n))$, its time complexity is $2^{\log(p(n))} = p(n)$. This means that M is a polynomial time transducer.

For $x, y \in \{0, 1\}$ it holds that $\neg x = 1 + (-1) \cdot x$, $x \vee y = x \cdot (1 + (-1) \cdot y) + y$ and $x \wedge y = x \cdot y$. So, by replacing Boolean gates with binary arithmetic circuits which compute the same value one can find a binary arithmetic circuit family $\overline{A}$ with size and height $4 \cdot p(n)$ which uses constant gates 1 and -1 and with $A_{|x|}(x) = B_{|x|}(x)$ for all $x \in \mathbb{N}$. Of course there is a polynomial time transducer M_A which describes $\overline{A}$.

We have to find an alternating polynomial time transducer M' which produces a family of full binary arithmetic circuits $\overline{C_{M'}}$ with $C_{M'(|x|)}(x) = A_{|x|}(x)$.

$M'(x)$ will first simulate $M_A(1^{|x|})$ and write down $A_{|x|}$ on its work tape. Then it will build a computation tree which resembles $A_{|x|}$. But since $A_{|x|}$, as an arithmetic circuit, need not be a tree whereas $C_{M'(x)}$ has to be a tree, some modifications have to be made:

Every sub-circuit of $A_{|x|}$ whose result is used by more than one gate will be replicated once for each usage of its result. This replication will start at the root gate.

The resulting binary arithmetic circuit $C_{M'(x)}$ can be of exponential size, but its height is identical to that of $A_{|x|}$. It can easily be padded so that it is fully balanced.

Since the height of $T_{M'}(|x|) = C_{M'(|x|)}$ is $4 \cdot p(|x|)$, its size is at most $2^{4 \cdot p(|x|)}$. Since $C_{M'(|x|)}(x) = A_{|x|}(x) = B_{|x|}(x)$, we know that $\overline{C_{M'}}$ decides L. This shows $L \in \mathrm{U_{FBT}\text{-}AC\text{-}SIZE\text{-}DEPTH}(2^{n^{O(1)}}, n^{O(1)})$. $\square$

Lemma 3.2. *Let $f \in \#\mathrm{APTIME}$. Then there exists a polynomial r such that for every polynomial q with $q \geq r$ there exists a circuit preserving APTIME TM M_q with computation trees of depth exactly $q(|x|)$, for which holds: $f(x) = \#(M_q, x)$.*

Proof. Let $f \in \#\mathrm{APTIME}$. We know $\#\mathrm{PSPACE} = \#\mathrm{APTIME}$. So, there is an NPSPACE TM M_f which shows $f \in \#\mathrm{PSPACE}$. We go full circle and use the proof of Proposition 2.1 to show that there exists an APTM M' which depends on M_f with $f(x) = \#(M', x)$ for all $x \in \mathbb{N}$. But we enhance the proof a little bit thus showing that M' produces completely balanced binary computation trees:

Let x be an input to M_f of length n. Without changing the number of accepting paths of M_f on x, we can pad all computations so they are of the same length $2^{p(n)}$ for some polynomial $p(n)$; we can also assume there is a unique accepting configuration. Consider the following alternating algorithm, $\mathrm{reach}(C, D, K)$, which accepts if and only if configuration D is reachable from configuration C in exactly 2^k steps.

```
function reach($C, D, k$);
begin
  if $k = 0$
    then
      if $D$ is reachable from $C$ in one step
        then *accept*
        else *reject*
    else
      $\bigvee E[\mathrm{reach}(C, E, k-1) \wedge \mathrm{reach}(E, D, k-1)]$
end
```

The notation $\bigvee E$ means existentially choose a configuration E. This existential branching can be done with a complete binary tree. If the number of configurations is not a power of 2, we can pad with non-reachable configurations. The notation $\wedge$ is a binary operator meaning universally choose one of its operands. So, one call of the function $\mathrm{reach}(C, D, k)$ yields a completely balanced computation tree. The recursive call "$[\mathrm{reach}(C, E, k-1) \wedge \mathrm{reach}(E, D, k-1)]$" hangs two complete trees of the same size on each leaf of the first tree. This continues until $k = 0$ and the whole computation tree is completely balanced.

Since the construction of the computation tree does not depend on the input, but only on the input length, all computation trees for inputs of the same length are equal.

The alternating algorithm that simulates M_f on input x is simply the call $\mathrm{reach}(\mathit{init}, \mathit{acc}, p(n))$, where *init* is the initial configuration of M_f on input x, and *acc* is the unique accepting configuration.

Since it can be shown by induction over k that the number of computation paths from C to D of length exactly 2^k equals the number of accepting computation trees of $\mathrm{reach}(C, D, k)$, the number of accepting computations of M_f on input x equals the number of accepting computation trees of $\mathrm{reach}(\mathit{init}, \mathit{acc}, p(n))$.

So, $M'(x)$ computes $\mathrm{reach}(\mathit{init}, \mathit{acc}, p(n))$. Let $h \colon \mathbb{N} \to \mathbb{N}$ be the function which describes the height of the computation tree depending on the input length. We can find a polynomial which is an upper bound for h. This polynomial will be our desired r. The TMs M_q for polynomials $q \geq r$ will, on input x, compute $q(|x|)$ and simulate $M'(x)$ while counting the height of the computation tree. When $M'(x)$ has finished at height $h(|x|)$ the computation tree will be padded by $q(|x|) - h(|x|)$ steps while not changing the number of accepted paths. $\square$

Our main result now gives a characterization of FPSPACE in terms of counting classes, arithmetic circuits, straight-line programs, matrix programs and leaf languages. The statement of Theorem 1.2 from the Introduction is equality $(1 = 8)$ of the following theorem.

Theorem 3.3. *Let Σ be the set of all 3×3 matrices over $\{-1, 0, 1\}$ and $\mathfrak{F} \colon \Sigma^* \to \mathbb{Z}$ with*

$$\mathfrak{F}(N_1, \ldots, N_n) =_{\mathrm{def}} \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \cdot \left(\prod_{i=1}^{n} N_i \right) \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

The following complexity classes are equal:

1) FPSPACE

2) Gap-PSPACE

3) #PSPACE − #PSPACE

4) #APTIME − #APTIME

5) $\mathrm{U_{FBT}}$-AC-SIZE-DEPTH$(2^{n^{O(1)}}, n^{O(1)})$

6) $\mathrm{U_{FBT}}$-SLP-SIZE-REG$(2^{n^{O(1)}}, 3)$

7) $\mathrm{U_{FBT}}$-MP-SIZE-DIM$(2^{n^{O(1)}}, 3)$

8) $\mathfrak{F}$-FP

Proof. ($1 \subseteq 3$): Let $f \in$ FPSPACE. We have to find two NPTMs M_1, M_2 with $f(x) = \#_p(M_1, x) - \#_p(M_2, x)$ for all $x \in \mathbb{N}$.

Let M_f be the PSPACE transducer which computes f. We can easily find two PSPACE transducers M_f^+ and M_f^- which simulate M_f and compute $\max\{f(x), 0\}$ and $\max\{-f(x), 0\}$ respectively. Both machines compute functions in FPSPACE$_+$ and it holds that $f(x) = M_f^+(x) - M_f^-(x)$.

This means FPSPACE $\subseteq$ FPSPACE$_+$ $-$ FPSPACE$_+$. But since we know that FPSPACE$_+$ $=$ #PSPACE, it follows FPSPACE $\subseteq$ #PSPACE $-$ #PSPACE.

($2 = 3$): "$\supseteq$": Let $f \in$ Gap-PSPACE. Let M_f be the NPSPACE machine which shows $f \in$ Gap-PSPACE. Then $f(x) = acc(M_f, x) - rej(M_f, x)$.

By defining the NPSPACE-TM M_f^+ as M_f we get $acc(M_f^+, x) = acc(M_f, x)$. We define the NPSPACE-TM M_f^- as follows: $M_f^-(x)$ simulates $M_f(x)$ and inverses acceptance and rejection. This means $acc(M_f^-, x) = rej(M_f, x)$.

But now we have $f(x) = acc(M_f^+, x) - acc(M_f^-, x)$, which gives us $f \in$ #PSPACE $-$ #PSPACE.

"$\subseteq$": Let $f \in$ #PSPACE $-$ #PSPACE. Then there exist two NPSPACE-TMs M_1 and M_2 with $f = acc(M_1, x) - acc(M_2, x)$. We define a new NPSPACE-TM M_f which behaves the following way on input x: Firstly, it forks in two branches. On the first it simulates $M_1(x)$ and if M_1 accepts, M_f accepts, too. If M_1 rejects, M_f forks in two paths; on the first it accepts and on the second it rejects.

On the second branch of the first forking M_2 will be simulated. If M_2 accepts, M_f forks in two paths; on the first it accepts and on the second it rejects. If M_2 rejects, M_f rejects, too.

It is easy to see that $acc(M_f, x) - rej(M_f, x) = acc(M_1, x) - acc(M_2, x)$. But this means $f \in$ Gap-PSPACE.

($3 = 4$): Immediate from Proposition 2.1.

($4 \subseteq 5$): Let $f \in$ #APTIME $-$ #APTIME. So, there exist two APTIME TMs M_1 and M_2 with $f(x) = \#(M_1, x) - \#(M_2, x)$ for all $x \in \Sigma^*$. Let M_1' and M_2' be two circuit preserving APTIME TMs which both produce computation trees of height $q(|x|)$ for some polynomial q and for which holds $\#(M_1, x) = \#(M_1', x)$ and $\#(M_2, x) = \#(M_2', x)$. Lemma 3.2 assures that such TMs exist.

The computation trees $T_{M_1'(x)}$ and $T_{M_2'(x)}$ can be viewed as arithmetic circuits which output $\#(M_1', x)$ and $\#(M_2', x)$, respectively. We will now build a new circuit preserving transducer M' with computation trees of height $q(|x|) + 2$, which only produces constant arithmetic circuits and with $C_{M'(x)} = \#(M_1', x) - \#(M_2', x)$:

$M'(x)$ forks existential. On both paths $M'(x)$ then forks universal. On path 1, $M_1'(x)$ will be simulated. On path 2, a completely balanced binary computation tree of height $q(|x|)$ with only universal forks will be built and on all paths of this subtree $M'(x)$ will output 1. We will call the arithmetic circuit without input corresponding to this subtree C_{S_1}. On path 3, a completely balanced binary computation tree of height $q(|x|)$ with only existential forks will be built and on its first path $M'(x)$ will output -1. On all other paths of the subtree 0 will be output. We will call the arithmetic circuit without input corresponding to this subtree C_{S_2}. On path 4, $M_2'(x)$ will be simulated. It is easy to see that $C_{S_1} = 1$ and $C_{S_2} = -1$. This leads to $C_{M'(x)} = (C_{M_1'(x)} * 1) + (-1 * C_{M_2'(x)}) = \#(M_1', x) - \#(M_2', x)$.

We have to find an alternating polynomial time transducer M which produces only full binary computation trees and which describes for each input x an arithmetic circuit $C_{|x|}$ (only dependent on the input length) with $C_{|x|}(x) = \#(M'_1, x) - \#(M'_2, x)$.

M' is circuit preserving, which means that its computation trees are only dependent on the input length—apart from the leaves, which depend on the input. But the only leaves which really depend on the input are the ones of the subtrees $T_{M'_1(x)}$ and $T_{M'_2(x)}$. Each of those leaves can be computed in deterministic polynomial time. So, we can find a polynomial time transducer M'_p which, for input (x, n) with $1 \leq n \leq 2^{q(|x|)+2}$, computes the output of path n of $M'(x)$. But, according to Lemma 3.1, we can find an arithmetic polynomial time transducer M'_c which produces an arithmetic circuit that computes $M'_p(x, n)$.

So, $M(x)$ simulates $M'(x)$ and, after finishing path n of its computation tree it simulates $M'_c(x, n)$. With this simulation another arithmetic circuit will replace the leaf n of $C_{M'(x)}$. But that circuit has a virtual input of (x, n). If one of its input leaves refers to a bit of the n part of its input, $M(x)$ has to replace that input gate by a constant gate which equals the bit of n.

After all the trees of the $M'_p(x, n)$ have been blown up to equal size, the resulting $T_{M(x)}$ depends solely on the input length, is a full binary computation tree, the output on its leaves will consist of $\{-1, 0, 1, 2, 3, \ldots, |x|+1\}$ and for its corresponding arithmetic circuit we have $C_{|x|}(x) = \#(M'_1, x) - \#(M'_2, x)$.

We conclude that $f \in \text{U}_{\text{FBT}}\text{-AC-SIZE-DEPTH}(2^{n^{O(1)}}, n^{O(1)})$.

$(5 \subseteq 6)$: Let $f \in \text{U}_{\text{FBT}}\text{-AC-SIZE-DEPTH}(2^{n^{O(1)}}, n^{O(1)})$. Then there is an alternating polynomial time transducer M which only produces fully balanced computation trees and with $C_{M(|x|)}(x) = f(x)$.

It is known ([BOC92], see also [Vol99a], Theorem 5.15) that the computation of an arithmetic circuit over the integers can be simulated by a straight-line program with only 3 registers. Let $P_{|x|}$ be the straight-line program that simulates the arithmetic circuit $C_{M(|x|)}$. We will now show that there is a nondeterministic polynomial time transducer M_S which, on input x, computes on each path one instruction of $P_{|x|}$, so that on the leaves of the computation tree of M_S the complete straight-line program—in correct order—will be found. We will say $M_S(x)$ *describes* $P_{|x|}$.

We will rely on the proof of Theorem 5.15 in [Vol99a]. There, following Ben-Or and Cleve [BOC92], the notion of *offset* is introduced: Let C be an arithmetic gate and $\{i, j, k\} = \{1, 2, 3\}$ be the set of registers. We want to find a straight-line program which offsets register R_j by $R_i * f_C$ and a program which offsets register R_j by $-R_i * f_C$. The proof shows that this is possible. The programs look like this:

If C is an input gate, the programs are $R_j \leftarrow R_j + f_C * R_i$ and $R_j \leftarrow R_j - f_C * R_i$, respectively. If C has predecessors D and E and $f_C = f_D + f_E$ or $f_C = f_D * f_E$, then we can construct the programs for f_C under the assumptions that we have straight-line programs which do the job for f_D and f_E:

Existential gate

Offset R_j by $R_i * f_C$:

1) offset R_j by $R_i * f_D$;

2) offset R_j by $R_i * f_E$;

3) offset R_j by $R_i * f_E$;

4) offset R_j by $-R_i * f_E$.

Offset R_j by $-R_i * f_C$:

1) offset R_j by $-R_i * f_D$;

2) offset R_j by $-R_i * f_E$;

3) offset R_j by $R_i * f_E$;

4) offset R_j by $-R_i * f_E$.

The last 2 subprograms in both programs are dummy programs to obtain a number of 4 subprograms.

Universal gate

Offset R_j by $R_i * f_C$:

1) offset R_j by $-R_k * f_E$;

2) offset R_k by $R_i * f_D$;

3) offset R_j by $R_k * f_E$;

4) offset R_k by $-R_i * f_D$.

Offset R_j by $-R_i * f_C$:

1) offset R_j by $R_k * f_E$;

2) offset R_k by $R_i * f_D$;

3) offset R_j by $-R_k * f_E$;

4) offset R_k by $-R_i * f_D$.

We see that each of the programs uses exactly 4 subprograms and in each subprogram the result of only one arithmetic gate is used. So, if we have a completely balanced arithmetic circuit of depth d—which is true for $C_{|x|}$—, the resulting straight-line program will consist of exactly 4^d instructions.

We will use the notion of *configurations*. The information needed to express the configuration on a certain computation step is the contents of the work tapes, the positions of the heads and the state of the transducer. For a transducer N, input x, and configuration c we will call $N(x,c)$ the computation of N on input x which starts in configuration c. It yields the computation tree $T_{N(x,c)}$.

Let c be a configuration reached by M on input x. Let the state of M at configuration c be universal resp. existential or an end state. $M(x,c)$ builds a computation tree $T_{M(x,c)}$ which can be expressed as an arithmetic circuit as shown. Let $G(x,c)$ be the root gate of $T_{M(x,c)}$. We will now prove by induction over the height h of $T_{M(x,c)}$ that there exists an nondeterministic polynomial time transducer M'_s which can describe the straight-line program which offsets register R_i by $R_j * f_{G(x,c)}$ and the straight-line program which offsets register R_i by $-R_j * f_{G(x,c)}$ for $i \neq j$.

$h = 0$: The root gate $G(x,c)$ is the only gate of $T_{M(x,c)}$; this means that the computation of M is in an end state. There, $M(x,c)$ outputs a value $f_{G(x,c)} \in \{-1, 0, 1, 2, \dots, |x| + 1\}$. If the straight-line program to be described is "Offset R_j by $R_i * f_{G(x,c)}$", then M'_S outputs the straight-line instruction "$R_j \leftarrow R_j + f_{G(x,c)} * R_i$" and if the program was "Offset R_j by $-R_i * f_{G(x,c)}$" then "$R_j \leftarrow R_j - f_{G(x,c)} * R_i$" is written.

$h \to h+1$: $G(x,c)$ has two successor gates $D(x,c_1)$ and $E(x,c_2)$ where c_1 and c_2 are the configurations reached by $M(x,c)$ while computing the left resp. right side of the branching at gate $G(x,c)$ until the new branching points $D(x,c_1)$ resp. $E(x,c_2)$ are reached. Let $G(x,c)$ wlog. be a universal gate and the straight-line program to be described wlog. be "Offset R_j by $R_i * f_{G(x,c)}$".

$M'_S(x,c)$ forks in 4 branches (since we want a binary tree this is done in two steps) and describes the following straight-line programs on the branches:

1) offset R_j by $-R_k * f_E(x, c_2)$;

2) offset R_k by $R_i * f_D(x, c_1)$;

3) offset R_j by $R_k * f_E(x, c_2)$;

4) offset R_k by $-R_i * f_D(x, c_1)$.

This is done by simulating the left path of $M(x, c)$ for branches 2) and 4) and the right path for branches 1) and 3) until the gates $D(x, c_1)$ resp. $E(x, c_2)$ are reached. By the induction hypothesis we know that the straight-line programs in 1) – 4) can be described by M'_S.

The machine M_S on input x forks in two branches. On the first branch a full binary computation tree of height d will be spanned. On its leaves will be written dummy instructions, e. g. $R_1 \leftarrow R_1 + 0 * R_1$. Only on its last two leaves the following instructions will be written:

1) $R_2 \leftarrow R_2 + 1 * R_1$

2) $R_1 \leftarrow R_1 - 1 * R_2$

On the second branch of the first forking, $M_S(x)$ will simulate $M_A(x)$ until the first universal resp. existential state is reached. Let the configuration of M_A at this state be c_A. Now M'_S will describe the straight-line program "Offset register R_1 by $R_2 * f_{G(x,c_A)}$".

Thus, $M_S(x)$ describes the program $P_{|x|}$, and, since $\overline{P}$ computes f, we conclude that $f \in$ $\mathrm{U_{FBT}}$-SLP-SIZE-REG$(2^{n^{O(1)}}, 3)$.

($6 \subseteq 7$): In Theorem 5.35 of [Vol99a] it was proved that each instruction of a straight-line program can easily (in polynomial time) be replaced by a matrix such that the resulting matrix program is computationally identical to the straight-line program. The number of used registers in the straight-line program will be the dimension of the matrices. Refer to [Vol99a] for details.

($7 \subseteq 8$): Let $f \in$ $\mathrm{U_{FBT}}$-MP-SIZE-DIM$(2^{n^{O(1)}}, 3)$. Then, there is a nondeterministic polynomial time transducer M which describes a family of matrix programs which computes f. Let h be the function which computes the number of leaves of $T_{M(x)}$ and let g be the function which, on input (x, n), computes the matrix computed on path n of $T_{M(x)}$, where the variables x_i are replaced by the i-th bit of x. Since both these functions are in FP, $f \in \mathfrak{F}$-FP.

($8 \subseteq 1$): Let $f \in F$-FP. This means, there are two polynomial-time computable functions $g\colon \Sigma^* \times \mathbb{N} \to \Sigma$ and $h\colon \Sigma^* \to \mathbb{N}$ such that for all x, $f(x) = \mathfrak{F}(g(x, 1), \ldots, g(x, h(x)))$.

The matrices $g(x, i)$ can easily be computed in polynomial space. We have to prove that the result

$$f(x) = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \cdot \left(\prod_{i=1}^{h(x)} g(x, i) \right) \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

can be computed in polynomial space, too. Our problem is the fact that $h(x)$ can be exponential in $|x|$ and this means that not only there are exponentially many matrices $g(x, i)$—which cannot all be stored in polynomial space—, but also the entries of the matrix $\prod_{i=1}^{h(x)} g(x, i)$ can grow exponential in $h(x)$ which gives them an exponential length in $|x|$.

But these difficulties can be overcome by a method used, e.g., in the proof of #APTIME $\subseteq$ FPSPACE in [Lad89]:

Let $\mathit{prod}(x, a, l) = \prod_{i=a}^{\min\{a+2^l-1, h(x)\}} g(x, i)$; then we have

$$f(x) = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \cdot \mathit{prod}(x, 1, |h(x)|) \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}.$$

Furthermore we can use the following fact to find a recursive procedure to compute $f(x)$: $\mathit{prod}(x, a, l+1) = \mathit{prod}(x, a, l) \cdot \mathit{prod}(x, a+2^l, l)$.

We are still not able to write down the partial products $\mathit{prod}(x, a, l)$ because of their size, but we will define a recursive procedure $\mathit{bit}(i, j, k, x, a, l)$ which computes the k-th bit of entry (i, j) of the matrix $\mathit{prod}(x, a, l)$:

```
function $\mathit{bit}(i, j, k, x, a, l)$;
begin
  if $l = 0$
    then
      return the $k$-th bit of entry $(i, j)$ of matrix $g(x, a)$
    else
      return the $k$-th bit of entry $(i, j)$ of matrix $\mathit{prod}(x, a, l-1) \cdot \mathit{prod}(x, a+2^{l-1}, l-1)$
end
```

Since there are well known algorithms which compute addition and multiplication in logarithmic space we can compute the matrix product of two—in $|x|$—exponentially sized matrices in space polynomial in $|x|$. If we want a certain bit of the output we just have to run this algorithm and discard its output until we arrive at the desired output bit. Whenever the algorithm asks for a bit of its input we make a recursive call to *bit*.

The depth of the recursion is $|h(x)|$, which is polynomial in $|x|$. On each step of the recursion we need space logarithmic in the size of matrices $\mathit{prod}(x, b, m)$ with $0 \leq m \leq |h(x)|$ which again is polynomial in $|x|$. Hence the computation of $\mathit{bit}(i, j, k, x, a, l)$ can be done in polynomial space. So, since the desired result $f(x)$ is the entry $(1, 1)$ of the matrix $\mathit{prod}(x, 1, |h(x)|)$, we can output it by successive calls to $\mathit{bit}(1, 1, k, x, 1, |h(x)|)$. □

Let C be a Boolean circuit. C defines the word over $\{0, 1\}$ whose ith bit is given by the output of C on input i. (In this way, actually only words whose length is a power of 2 can be defined by circuits; however, there are different ways to handle arbitrary word lengths, for technical details see, e.g., [Vei98].) Say that the succinct version of a language L is the set of all those Boolean circuits that define words in L.

Helmut Veith showed that if a complexity class $\mathcal{C}$ is characterized by a leaf language L, then the succinct version of L is complete for $\mathcal{C}$ under First-order projections [Vei98]. First-oder projections [Imm99] are a uniform version of Valiant's projection reductions [Val82]. Thus, we conclude:

Corollary 3.4. *The succinct encoding of the problem, given a sequence of 3×3 matrices over $\{-1, 0, 1\}$, to determine if the entry in the upper left corner of the product is non-zero, is complete for* PSPACE *under first-order projections.*

Proof. It is immediate from Theorem 3.3 that the function $\mathfrak{F}$ in its decision version ("Is the upper left entry in the product of the input matrices non-zero?") is a leaf language for PSPACE. Thus, our corollary follows using Veith's result. □

4 Proof of Theorem 1.1

Theorem 1.1. *There is a finite automaton M with output such that for the function f_M computed by M, we have: f_M-FP = FPSPACE.*

Proof. Let $f \in$ FPSPACE. Then, the set of all pairs (x, i) such that the i-th bit of $f(x)$ is on is in PSPACE. Express this PSPACE language using NPTM M and some regular leaf language B with non-solvable syntactic monoid. Now define a NPTM M' operating as follows: On input x, M' first branches for all values of i in a certain exponential range, and then simulates M on input (x, i). Also, we have to make sure that the blocks for different values of i are separated by a certain symbol # in the leaf string. Consider the finite automaton M'' reading the leaf string of M' and operating as follows: While reading leaf symbols within a block for some i, it simulates a finite automaton M''' for B. When M'' encounters a block marker #, it outputs 1 iff M''' is in a final state, and 0 otherwise. In the next block, M'' resumes the simulation of M''' in its initial state. Thus, M'' outputs a binary value when it reads a # and produces no output for other leaves. The outputs at a block marker, however, are exactly the bits of the value $f(x)$. □

Acknowledgment. We thank Sven Kosub, München, for enlightening discussions at several stages during the development of this paper. We also thank Klaus W. Wagner, Würzburg, for pointing out Theorem 1.1.

References

[Bar89] D. A. Mix Barrington. Bounded-width polynomial size branching programs recognize exactly those languages in NC^1. *Journal of Computer and System Sciences*, 38:150–164, 1989.

[BCS92] D. P. Bovet, P. Crescenzi, and R. Silvestri. A uniform approach to define complexity classes. *Theoretical Computer Science*, 104:263–283, 1992.

[BOC92] M. Ben-Or and R. Cleve. Computing algebraic formulas using a constant number of registers. *SIAM Journal on Computing*, 21:54–58, 1992.

[CF91] J.-Y. Cai and M. Furst. PSPACE survives constant-width bottlenecks. *International Journal of Foundations of Computer Science*, 2:67–76, 1991.

[CMTV98] H. Caussinus, P. McKenzie, D. Thérien, and H. Vollmer. Nondeterministic NC^1 computation. *Journal of Computer and System Sciences*, 57:200–212, 1998.

[GW83] H. Galperin and A. Wigderson. Succinct represantation of graphs. *Information and Control*, 56:183–198, 1983.

[HLS$^+$93] U. Hertrampf, C. Lautemann, T. Schwentick, H. Vollmer, and K. W. Wagner. On the power of polynomial time bit-reductions. In *Proceedings 8th Structure in Complexity Theory*, pages 200–207, 1993.

[Imm99] N. Immerman. *Descriptive Complexity.* Graduate Texts in Computer Science. Springer Verlag, New York, 1999.

[KSV00] S. Kosub, H. Schmitz, and H. Vollmer. Uniform characterizations of complexity classes of functions. *International Journal of Foundations of Computer Science*, 11(4):525–551, 2000.

[Lad89] R. Ladner. Polynomial space counting problems. *SIAM Journal on Computing*, 18(6):1087–1097, 1989.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, MA, 1994.

[Val82] L. Valiant. Reducibility by algebraic projections. *L'enseignement mathématique*, 28:253–268, 1982.

[Vei98] H. Veith. Succinct representation, leaf languages, and projection reductions. *Information & Computation*, 142:207–236, 1998.

[Ver93] N. K. Vereshchagin. Relativizable and non-relativizable theorems in the polynomial theory of algorithms. *Izvestija Rossijskoj Akademii Nauk*, 57:51–90, 1993. In Russian.

[Vol99a] H. Vollmer. *Introduction to Circuit Complexity – A Uniform Approach.* Texts in Theoretical Computer Science. Springer Verlag, Berlin Heidelberg, 1999.

[Vol99b] H. Vollmer. Uniform characterizations of complexity classes. *Complexity Theory Column 23, ACM-SIGACT News*, 30(1):17–27, 1999.

[VT89] H. Venkateswaran and M. Tompa. A new pebble game that characterizes parallel complexity classes. *SIAM J. on Computing*, 18:533–549, 1989.