On the Autoreducibility of Functions*

Piotr Faliszewski
pfali@cs.rochester.edu

Mitsunori Ogihara
ogihara@cs.rochester.edu

Univ. of Rochester Comp. Sci. Dept. Technical Report TR-2007-912
January 23, 2007

Abstract

This paper studies the notions of autoreducibility and length-decreasing self-reducibility of functions and languages. Recently Glaßer *et al.* have shown that for many classes $\mathcal{C}$, including PSPACE and NP, it holds that all nontrivial complete languages are polynomial-time many-one autoreducible. In contrast, this paper shows that for many classes $\mathcal{C}$ such that $\mathrm{P} \subseteq \mathcal{C}$ (e.g., PSPACE and NP) some complete languages in $\mathcal{C}$ are not polynomial-time length-decreasing self-reducible unless $\mathcal{C} \subseteq \mathrm{P}$, and for classes $\mathcal{C}$ such that $\mathrm{L} \subseteq \mathcal{C} \subseteq \mathrm{P}$ (e.g., P and NL) some complete languages in $\mathcal{C}$ are not logarithmic-space length-decreasing self-reducible unless $\mathcal{C} \subseteq \mathrm{L}$.

This paper also shows that contrast between autoreducibility and length-decreasing self-reducibility for the case of functions. In particular, the paper shows that many function complexity classes $\mathcal{FC}$ (including well-studied #P, SpanP, and GapP and not-so-well-studied but highly natural #PE and TotP) have the property that all complete functions in $\mathcal{FC}$ are polynomial-time Turing-autoreducible. For #P and TotP, the autoreductions can be made to be polynomial-time one-Turing (one query per input).

These results show that, under reasonable assumptions, the notions of length-decreasing self-reducibility and autoreducibility differ both on complete languages and on complete functions. In a similar vein, this paper shows that under reasonable assumptions autoreducibility and random-self-reducibility differ with respect to functions.

1 Introduction

Self-reducibility [MP79, Sch76] and autoreducibility [Lad73, Amb83] are among the most frequently used central concepts in complexity theory. Intuitively, these notions refer to the situations in which the membership question about a word in a language can be answered by asking the membership question in the same language about other words. While autoreducibility essentially permits querying about *any word other than the input* (within a certain resource constraint), self-reducibility permits querying only those words preceding the input with respect to some partial order (the exact definition of the partial order changes the characteristics of the self-reducibility). It is well-known that the NP-complete problem SAT possesses such a property: Given a non-trivial formula φ as input, one can decide whether φ is satisfiable by asking whether at least one of the two formulas constructed by fixing the value of the first variable of φ is satisfiable. This is called the *disjunctive-self-reducibility* of SAT. Self-reducibility and autoreducibility apply to functions as well. For example, a #P-complete function #SAT, which on a propositional formula φ as input returns the number of satisfying truth assignments for φ, can be computed using the recursion:

*This technical report subsumes URCS-TR 2005-874.

$\#\mathrm{SAT}(\varphi) = \#\mathrm{SAT}(\varphi_0) + \#\mathrm{SAT}(\varphi_1)$, where φ_b is the formula φ with its first variable set to b. We will call this type of autoreducibility in which the value of a function at an input x is computed by a linear combination of the value of the function at some other inputs plus an additive factor *affine autoreducibility.*

A rich theory of self-reducibility and autoreducibility has been established by studying the structure of the reductions, that is, how "easier" the queries should be and how powerful the underlying computation is. The theory encompasses such concepts as coherence [Yao90], log-self-reducibility [Bal90], random-self-reducibility [AFK89], and word-decreasing self-reducibility [Bal90].[1] Much work in the area of autoreducibility has been done on random-self-reducibility [Yao90, AFK89, FKN90, FFLS92, FF91]. Intuitively, f is random-self-reducible if we can deduce the value of f at x from the values $f(x_1), \ldots, f(x_k)$, where x_i's are chosen in such a way that each x_i looks as if it was drawn from a distribution that only depends on $|x|$ and i; the x_i's are not necessarily independent, and typically they are not. Coherence is intuitively the probabilistic version of autoreducibility in which the queries can be generated with access to randomness and the final assertion should be produced with a small error probability. Random-self-reducibility has a wide variety of applications in other topics, including average-case complexity and program testing. For the former, it is known, for example, that if a function f is random-self-reducible then, it is "as hard on the average as" it is in the worst case. For the latter, it is known that if a function f is randomly-self-reducible and we have a deterministic program that computes the function correctly for all but a small fraction of inputs, then the deterministic program can be modified to a probabilistic one that computes the correct value with high probability for all inputs [Lip91].

Deterministic versions of self-reducibility and autoreducibility of functions did not receive as much attention in the literature as their randomized counterparts, with a notable exception of the paper by Pagourtzis and Zachos [PZ06] which studies self-reductions of TotP and #PE functions. (TotP is the class of functions that count the total number of computation paths of NP machines and #PE is the set of #P functions f for which one can determine in polynomial time whether $f(x) > 0$.) Here, in this paper, we attempt to fill the gap in the literature regarding deterministic self-reducibility and autoreducibility of functions.

One of the reasons to study autoreducibility of functions, apart from the natural curiosity, is that it might lead to natural separations of complexity classes. Buhrman *et al.* [BFvMT00] have already shown that using autoreducibility one can separate certain classes (though, there are also other techniques to achieve some of their separations) and in the case of function classes this attack might be very attractive as well. In particular, in this paper we show that not all $\mathrm{FP_t}$-complete functions are parsimoniously autoreducible (see Section 2.4; $\mathrm{FP_t}$ is the set of total FP functions), but all #P-complete functions are very close to having such autoreductions. Also, recent results on the many-one autoreducibility of NP-complete languages [GOP$^+$05] raise our hope to be able to show that all #P-complete functions are parsimoniously autoreducible, and thus to separate #P and $\mathrm{FP_t}$. We study the autoreducibility of complete functions in Section 4.

Like the language SAT and the function #SAT, many concrete complete sets and functions are known to be self-reducible. In many cases their self-reductions are *length-decreasing* in the sense

[1] Some of these variants of "self-reducibility" go beyond what the seminal papers of Meyer and Paterson [MP79] and of Schnorr [Sch76] intended to capture. In their work self-reducibility refers to situations in which autoreducibility is established according to a partial order that gives "short" downward chains. Such a short-downward-chain property does not exist for some of the variants, in particular, for random-self-reducibility and word-decreasing self-reducibility. This lack seems to blur the distinction between "self-reducibility" and autoreducibility. All the self-reducibility notions we study are special cases of length-decreasing self-reducibility. Although the term "length-decreasing autoreducibility" adequately characterizes the property, we follow the convention and keep the word "self-reducibility" in the names.

that the query words are shorter than the input. It is easy to see that the self-reductions of SAT and #SAT presented above are indeed length-decreasing. The standard complete language QBF for PSPACE has a similar self-reduction in which the membership is queried about the formulas constructed by fixing the first variable to 0 and to 1. This self-reduction is length-decreasing too. It is implicit in the work of Pagourtzis and Zachos [PZ06] that, using a certain natural encoding of graphs, the function #PerfectMatching that given a graph as input returns the number of perfect matchings in it is also length-decreasing self-reducible.

It is natural to ask whether every complete problem or function for, e.g., PSPACE, NP, and #P, is indeed length-decreasing self-reducible. We prove that this is unlikely. For a wide variety of classes $\mathcal{C}$, including PSPACE and NP, it holds that if all complete sets for $\mathcal{C}$ are length-decreasing self-reducible then $\mathcal{C} \subseteq \mathrm{P}$. A similar result holds for #P and other function classes.

The above result about NP can be contrasted with the results about autoreducibility of NP-complete languages. Formally, a language A is autoreducible if it is accepted by a polynomial-time oracle Turing machine M such that M relative to oracle A accepts A, viz., $L(M^A) = A$, and on input x machine M never queries its oracle about x. Beigel and Feigenbaum [BF92] showed that all languages complete for NP with respect to polynomial-time Turing reductions are (Turing) autoreducible. More recently, Glaßer *et al.* [GOP$^+$05] showed that all NP-complete sets and all PSPACE-complete sets are many-one autoreducible. Thus, from our new result it follows that, e.g., under the assumption that P $\neq$ PSPACE there are PSPACE-complete languages that are autoreducible but not length-decreasing self-reducible.

Buhrman and Torenvliet [BT96] gave evidence that general self-reducibility (as defined by Meyer and Paterson in [MP79]) differs from autoreducibility on NP. Our result does not follow from their result, since the length-decreasing self-reducibility is a special case of the Meyer–Paterson self-reducibility. Indeed, our results subsume the result of Buhrman and Torenvliet.

While the notions of self-reducibility and autoreducibility have been diversified and well studied in the literature for languages,[2] the notions have not been well explored for functions. We will fill the gap both by generalizing the language-based notions and by introducing, based on the ideas of Pagourtzis and Zachos [PZ06], the aforementioned affine reduction type and their subtypes.

This paper is organized as follows. In Section 2 we give basic definitions that will be useful throughout the paper. In particular, we discuss reducibility notions and self-reducibility and autoreducibility for function classes. Section 3 presents our results regarding length-decreasing self-reducibility, namely, that for many natural classes it is not the case that all complete languages (complete functions) within these classes are length-decreasing self-reducible unless unlikely complexity class collapses occur. In Section 4 we turn our attention to autoreducibility of functions and prove several analogs of results known for the autoreducibility of complete languages. In particular, we show that many natural #P functions have autoreductions of a very simple form, and we argue how autoreducibility could be used to separate #P from FP_t. We conclude this section with a brief discussion of algebraic properties of complete functions. Section 5 concludes the paper with a few open problems.

2 Preliminaries

We assume the reader's familiarity with basic concepts in complexity theory. The reader may consult with such textbooks as Bovet and Crescenzi [BC93], Hemaspaandra and Ogihara [HO02],

[2] Köbler and Watanabe [KW98] use a very generic notion of self-reducibility, which can be applied beyond PSPACE (all length-decreasing self-reducible sets, and all sets self-reducible with respect to the Meyer–Paterson self-reducibility, are in PSPACE).

and Papadimitriou [Pap94] for definitions. Without loss of generality, we assume that all languages we consider are over the alphabet $\Sigma = \{0, 1\}$. We assume a pairing function that is both polynomial-time computable and polynomial-time decodable (for example, Cantor's pairing function). Let $\langle \cdot, \cdot \rangle$ be such a function. All polynomials we consider here have positive coefficients so for all $n \geq 0$ their value is nonnegative. We use the term "NP machine" to mean any polynomial time-bounded nondeterministic Turing machine.

2.1 Complexity Classes

Next we define complexity classes of our interest.

PSPACE is the set of all languages that are accepted by polynomial-space deterministic Turing machines. L and NL are respectively the set of languages that are accepted by logarithmic-space deterministic Turing machines and the set of languages accepted by logarithmic-space nondeterministic Turing machines.

A language L belongs to the class PP [Sim75, Gil77] if there exists a polynomial p and a polynomial-time decidable predicate R such that for all x,

$$x \in L \iff \|\{y \mid |y| = p(|x|) \wedge R(x, y)\}\| \geq 2^{p(|x|)-1}.$$

A language L belongs to $\mathrm{C_{=}P}$ [Sim75, Wag86] if and only if there is a polynomial p and a polynomial-time decidable predicate R such that for all x,

$$x \in L \iff \|\{y \mid |y| = p(|x|) \wedge R(x, y)\}\| = 2^{p(|x|)-1}.$$

Both PP and $\mathrm{C_{=}P}$ can be viewed as representing the concept of counting the number of accepting paths of NP machines, that is, PP asks if at least a half of the paths accept and $\mathrm{C_{=}P}$ asks whether exactly a half of the paths accept.

Fact 2.1 below, which is derived from the fact that $\mathrm{coNP} \subseteq \mathrm{C_{=}P}$ [Sim75] and the fact that $\mathrm{PP} \subseteq \mathrm{NP}^{\mathrm{C_{=}P}}$ [Tor91], is well known.

Fact 2.1 $\mathrm{P} = \mathrm{C_{=}P}$ *if and only if* $\mathrm{P} = \mathrm{PP}$.

The *rank* of a word x among some set of words A is the number of words in A that are lexicographically smaller than x. Without explicit specification, A by default is Σ^*. We denote the function that maps each x to its rank in Σ^* by $\mathrm{rank}(x)$.

A *Turing machine transducer* is a Turing machine with a special one-way infinite write-only tape called *output tape* that is accessed via a head moving only one way. We assume that the output of a Turing machine transducer is over the alphabet Σ and that the output head cannot be moved before writing in the tape cell at the current position. A Turing machine transducer M outputs a word w on input x if M on x accepts and w is the word written on the output tape.

A function $f : \Sigma^* \to \Sigma^*$ is *polynomial-time computable* if there exists a polynomial time-bounded Turing machine transducer M that computes f.[3] A function $f : \Sigma^* \to \Sigma^*$ is *logarithmic-space computable* if there exists a logarithmic-space Turing machine transducer M that computes f. Here the logarithmic bound is not enforced on the output tape. The class of all polynomial-time computable functions and the class of all logarithmic-space computable functions are denoted respectively by FP and FL. The sets of total FP and FL functions are denoted by $\mathrm{FP_t}$ and $\mathrm{FL_t}$.

A function $f : \Sigma^* \to \Sigma^*$ can be viewed, by way of an interpretation scheme, as a mapping from Σ^* to any finite or countably infinite set of objects, including the natural numbers $\mathbb{N}$, the integers

[3] Usually, a Turing machine transducer is considered to have an output if and only if it accepts. So, both FP and FL contain not only *total* functions but *partial* functions. We will consider only total functions here.

$\mathbb{Z}$, and the rationals $\mathbb{Q}$. Instead of fixing the interpretation to one scheme, we simply stipulate that Σ^* is viewed as a semiring. (We'll return to the issue of interpretation in Section 2.4.) With this stipulation, to mean that every function in a class of functions $\mathcal{FC}$ is polynomial-time computable, we write $\mathcal{FC} \subseteq \mathrm{FP}$ instead of $\mathcal{FC} = \mathrm{FP}$ even if it is the case that with respect to some typical interpretation $\mathrm{FP} \subseteq \mathcal{FC}$.

Next we define function classes #P [Val79], GapP [FFK94], SpanP [KST89], TotP [KPSZ98], and #PE [Pag01].

Let M be an NP machine. For each input x to M, define $\#\mathrm{acc}_M(x)$, $\#\mathrm{rej}_M(x)$, and $\#\mathrm{tot}_M(x)$, to be the number of accepting, rejecting, and all computation paths of M on x, respectively. Clearly, for all x it holds that $\#\mathrm{tot}_M(x) = \#\mathrm{acc}_M(x) + \#\mathrm{rej}_M(x)$.

Let f be a function from Σ^* to $\mathbb{N}$. We say that f belongs to #P if there is an NP machine M such that for all x, $f(x) = \#\mathrm{acc}_M(x)$. We say that f belongs to TotP if there is an NP machine M such that for all x, $f(x) = \#\mathrm{tot}_M(x) - 1$. (The subtraction of 1 is for allowing TotP functions to have 0 as their value.) We say that f belongs to #PE if $f \in \#\mathrm{P}$ and the language $\{x \mid f(x) > 0\}$ is in P.

A function $f : \Sigma^* \to \mathbb{Z}$ is in GapP if there is an NP machine M such that for all x, $f(x) = \#\mathrm{acc}_M(x) - \#\mathrm{rej}_M(x)$. A function $f : \Sigma^* \to \mathbb{N}$ is in SpanP if there is an NP Turing machine transducer M such that for all x, $f(x)$ is the number of distinct outputs of M on input x.

It follows directly from the definitions that, using the semiring $\mathbb{N}$ to interpret $\mathrm{FP_t}$ values, $\mathrm{FP_t} \subseteq \mathrm{TotP} \subseteq \#\mathrm{PE} \subseteq \#\mathrm{P} \subseteq \mathrm{SpanP}$ and that $\#\mathrm{P} \subseteq \mathrm{GapP}$. The following result, which we will use later, is due to Köbler, Schöning, and Torán.

Fact 2.2 ([KST89]) $\#\mathrm{P} = \mathrm{SpanP} \iff \mathrm{UP} = \mathrm{NP}$.

2.2 Self-reducibility and Autoreducibility Notions for Languages

As is standard $\leq_m^p$, $\leq_T^p$, $\leq_{tt}^p$, $\leq_m^{\log}$, $\leq_T^{\log}$, and $\leq_{tt}^{\log}$ respectively stand for polynomial-time many-one reducibility, polynomial-time Turing reducibility, polynomial-time truth-table reducibility, logspace many-one reducibility, logspace Turing reducibility, and logspace truth-table reducibility. Note that, as Ladner and Lynch showed [LL76], $\leq_T^{\log}$ is equivalent to $\leq_{tt}^{\log}$.

Definition 2.3 *A language A is autoreducible (more specifically, polynomial-time Turing autoreducible, or $\leq_T^p$ autoreducible) if there exists a polynomial-time oracle Turing machine M such that*

1. *$L(M^A) = A$ and*
2. *for all inputs x, M does not query its oracle about x.*

It is easy to see that SAT is autoreducible by the reduction presented in the introduction: On input formula φ, if φ does not contain a variable then accept or reject according to whether φ is true or false; otherwise, construct two formulas, φ_0 and φ_1, by setting the value of the first variable of φ to 0 and 1, respectively, ask the oracle about the membership of these formulas, and then accept if and only if the oracle answers positively to at least one of the two queries. In this reduction the queries are disjunctive and the query words are shorter than the input, so it is actually a disjunctive length-decreasing self-reduction.

Formally, we define length-decreasing self-reductions as follows:

Definition 2.4 *We say that a language A is length-decreasing self-reducible (more specifically, polynomial-time length-decreasing Turing self-reducible or $\leq_T^p$ length-decreasing self-reducible) if there exists a polynomial-time oracle Turing machine M witnessing that A is autoreducible with*

an additional property that for all x, each query word (if any) of M on input x is shorter than x, regardless of its oracle.

The logarithmic-space versions of the above two notions are defined simply by requiring that the machine M should run in logarithmic space. As in the standard definition by Ladner and Lynch [LL76], the logarithmic bound does not apply to the oracle tape, and thus, the queries can be of up to polynomial size. Since the logarithmic-space Turing (i.e., adaptive query) reduction is the same as the logarithmic-space truth-table (i.e., nonadaptive query) reduction, we will use the symbol $\leq_{tt}^{\log}$ instead of $\leq_{T}^{\log}$.

Let f be a function from $\mathbb{N}$ to $\mathbb{N}$. We say that f is *polynomially bounded* if there exists a polynomial p such that for all $n \geq 0$ it holds that $f(n) \leq p(n)$. We say that f is a *pad-length function* if for all $n \geq 0$ it holds that $f(n) > n$. We say that a pad-length function f is *logspace computable* if the mapping $1^n \mapsto 1^{f(n)}$ is computable in logarithmic space and *polynomial-time computable* if the mapping is computable in polynomial time. Note that every logspace computable (or polynomial-time computable) pad-length function is polynomially bounded.

Definition 2.5 *We say that a class $\mathcal{C}$ is closed under logspace padding (respectively, polynomial-time padding) if for every nontrivial (neither $\{0,1\}^*$ nor $\emptyset$) $A \in \mathcal{C}$ and for every logspace computable (respectively, polynomial-time computable) pad-length function f, the language*

$$A' = \{x10^m \mid x \in A \wedge 1 + |x| + m = f(|x|)\}$$

belongs to $\mathcal{C}$.

It is easy to see that L, NL, and P are closed under logspace padding and P, NP, PP, $C_{=}P$, and PSPACE are closed under both polynomial-time padding and logspace padding.

Lemma 2.6 *Let $\mathcal{C}$ be a language class that has $\leq_m^{\log}$-complete sets (respectively, $\leq_m^p$-complete sets) and is closed under logspace (respectively, polynomial-time) padding. Let f be an arbitrary logspace computable (respectively, polynomial-time computable) pad-length function. Let A be an arbitrary $\mathcal{C}$-complete set. Then $A' = \{x10^m \mid x \in A \wedge |x| + 1 + m = f(|x|)\}$ is $\mathcal{C}$-complete.*

Proof: We will prove only the logspace case. Let $\mathcal{C}$, f, and A be as in the hypothesis of the lemma. Since $\mathcal{C}$ is closed under logspace padding, $A' \in \mathcal{C}$. To show that A' is $\leq_m^{\log}$-hard for $\mathcal{C}$, define $g(x) = x10^{|f(x)|-|x|-1}$. Clearly, g is a many-one reduction from A to A'. Since f is a logspace computable pad-length function, g is logspace computable. Thus, A' is $\leq_m^{\log}$-hard for $\mathcal{C}$. This proves the lemma. ■

2.3 Reducibility Notions for Function Classes

Here we list the reducibility notions among functions to be used in this paper. All the reducibility notions defined in this section are deterministic. Recall that we only consider total functions here. The following definitions apply only to total functions and, as a result, we cannot speak of complete functions for classes that also contain partial functions.

A *function-oracle Turing machine* is a standard Turing machine with a special write-only query tape and with a special read-only *answer tape*. When the machine enters the query state, in a single unit time the contents of the query tape are erased and the oracle produces a word over Σ on the answer tape.

Definition 2.7 *We say that a function f is (polynomial-time) Turing reducible to a function g, denoted $f \leq^p_T g$, if there is a polynomial-time computable function-oracle Turing machine that computes f with g as the oracle.*

Defining the function version of the "many-one" reduction is not straightforward. Under the obvious restriction that the function-oracle Turing machine computing the reduction should not make more than one query to the oracle, one can consider various issues regarding how computation should be carried out before and after the query: whether the machine should supply the input without modification to the oracle, whether the machine should output the answer provided by the oracle, and if not, whether the machine should be allowed to look at the input during the post-computation phase.

The definition below, due to Krentel [Kre88], is the most flexible one.

Definition 2.8 ([Kre88]) *A function f is (polynomial-time) metric reducible to a function g, denoted by $f \leq^p_{1\text{-}T} g$, if there exist two polynomial-time computable functions ψ and φ such that for all x, $f(x) = \varphi(x, g(\psi(x)))$.*

The notation $f \leq^p_{1\text{-}T} g$ is different from the notation used in [Kre88]. Our notation is adopted from [WT92] and signifies that the machine computing the reduction has the full liberty in computation before and after making its single query.

Two more restrictive reduction types are Zankó's *many-one reducibility* [Zan91], in which post-computation should depend only on the oracle answer, and Simon's *parsimonious reducibility* [Sim75],[4] in which no post-computation is allowed.

Definition 2.9 ([Zan91, Sim75]) *A function f is (polynomial-time) many-one reducible to a function g, denoted $f \leq^p_m g$, if there exist two polynomial-time computable functions ψ and φ such that for all x, $f(x) = \varphi(g(\psi(x)))$.*

A many-one reduction is called parsimonious, $f \leq^p_{par} g$, if there exists a polynomial-time computable function ψ such that for all x, $f(x) = g(\psi(x))$.

Next we introduce the notion of reversible metric reducibility. Intuitively, a function f is reversible metric reducible to a function g if for each input x and for the query, y, for x, it holds not only that $f(x)$ can be computed from y and $g(y)$ in polynomial time but $g(y)$ can be computed from x and $f(x)$ in polynomial time.

Definition 2.10 *A function f is (polynomial-time) reversible metric reducible to a function g, denoted $f \leq^p_{1\text{-}T\text{-rev}} g$, if there exist polynomial-time functions φ, ψ, and ρ such that φ and ψ witness that $f \leq^p_{1\text{-}T} g$ and such that for all x, $\rho(x, f(x)) = g(\psi(x))$.*

We conclude this subsection with a quick discussion of some basic properties of the reductions that we are dealing with. As is both expected and very natural, all of the reductions presented here are transitive.

Proposition 2.11 *Each of $\leq^p_{par}$, $\leq^p_m$, $\leq^p_{1\text{-}T}$, $\leq^p_{1\text{-}T\text{-rev}}$, and $\leq^p_T$ is transitive.*

[4] Vollmer [Vol94] uses the term "functional many-one reducibility" to refer to the parsimonious reducibility.

Proof: We will only do a proof for the case of $\leq^p_{1\text{-}T\text{-rev}}$. The transitivity of the other reductions should be clear. Let f, g, and h be three functions such that $f \leq^p_{1\text{-}T\text{-rev}} g$ (via φ_1, ψ_1, and ρ_1) and $g \leq^p_{1\text{-}T\text{-rev}} h$ (via φ_2, ψ_2, and ρ_2). That is, for each x we have that

$$\begin{aligned} f(x) &= \varphi_1(x, g(\psi_1(x))), \\ \rho_1(x, f(x)) &= g(\psi_1(x)), \end{aligned}$$

and for each x we have that

$$\begin{aligned} g(x) &= \varphi_2(x, h(\psi_2(x))), \\ \rho_2(x, g(x)) &= h(\psi_2(x)). \end{aligned}$$

The goal is to define φ, ψ, and ρ that witness $f \leq^p_{1\text{-}T\text{-rev}} h$. We define φ and ψ as follows:

$$\begin{gathered} \text{for each } x,\ \psi(x) = \psi_2(\psi_1(x)), \\ \text{for each } x \text{ and for each } y,\ \varphi(x, y) = \varphi_1(x, \varphi_2(\psi_1(x), h(y))). \end{gathered}$$

It is clear that both ψ and φ are in $\mathrm{FP_t}$. Also, it is easy to check that $\varphi(x, \psi(x)) = f(x)$. It remains to show that we can compute $h(\psi(x))$ given x and $f(x)$. For each x and for each y, define $\rho(x, y)$ by:

$$\rho(x, y) = \rho_2(\psi_1(x), \rho_1(x, y)).$$

Clearly, $\rho \in \mathrm{FP_t}$. For all x, we have:

$$\begin{aligned} \rho(x, f(x)) &= \rho_2(\psi_1(x), \rho_1(x, f(x))) \\ &= \rho_2(\psi_1(x), g(\psi_1(x))) \\ &= h(\psi_2(\psi_1(x))) \\ &= h(\psi(x)). \end{aligned}$$

This completes the proof. ■

Our reduction types vary considerably in the amount of flexibility that they allow. In particular, the Turing reduction puts no restrictions on the number and nature of the queries that a reduction can make and the parsimonious reduction allows us to make only one query, without any post-computation. The following set of implications clearly follows from the definitions of the reducibilities:

1. if $f \leq^p_{par} g$ then $f \leq^p_m g$;
2. if $f \leq^p_m g$ then $f \leq^p_{1\text{-}T} g$;
3. if $f \leq^p_{1\text{-}T} g$ then $f \leq^p_T g$.

These implications cannot be replaced by equivalences. In each of the cases there are simple examples of functions f and g for which it holds that f reduces to g by the stronger reduction but does not by the weaker one.

Let $f(x) = x$ and $g(x) = 2x$. (We are considering x here to be a natural number and we view f and g as FP functions over the semiring of natural numbers.) It holds that $f \leq^p_m g$ but it clearly is not the case that $f \leq^p_{par} g$.

To show that metric reductions can be more powerful than Zankó's, let f be some $\mathrm{FP_t}$ function that is not constant and let g be a constant function. It holds that $f \leq^p_{1\text{-}T} g$, but it is not the case that $f \leq^p_m g$ as the post-computation in the reduction always gets the same input argument.

To show that Turing reductions are more powerful than metric reductions we recall that Ladner, Lynch, and Selman [LLS75] showed two languages, A and B, such that $A \leq_T^p B$ but B does not reduce to A in the truth-table fashion. Let χ_A and χ_B be characteristic functions of A and B. Since $A \leq_T^p B$, it holds that $\chi_A \leq_T^p \chi_B$. Also, a metric reduction from χ_A to χ_B would imply a truth-table reduction from A to B, which is impossible.

Perhaps the most peculiar reducibility that we use is the reversible metric reduction. In particular, it is interesting to compare its power to Zankó's many-one reducibility. It is easy to provide an example of two functions f and g such that $f \leq_{1\text{-}T\text{-rev}}^p g$ but it is not the case that $f \leq_m^p g$. One, as in the case of general metric reductions, simply takes g to be a constant function and f to be an $\mathrm{FP_t}$ function that is not constant. Interestingly, assuming $\mathrm{P} \neq \mathrm{NP}$, one can construct two recursive functions f and g such that $f \leq_m^p g$ and $f \not\leq_{1\text{-}T\text{-rev}}^p g$. (In fact, if f and g are allowed to be non-recursive, a pair (f, g) satisfying $f \leq_m^p g$ and $f \not\leq_{1\text{-}T\text{-rev}}^p g$ exists unconditionally.)

Lemma 2.12 *There exist two recursive functions, f and g, such that (i) $f \leq_m^p g$ and (ii) $f \not\leq_{1\text{-}T\text{-rev}}^p g$ unless* $\mathrm{P} = \mathrm{NP}$.

Proof: We define $f(x) = \chi_{\mathrm{SAT}}(x)$ to be the characteristic function of SAT and we define g as follows.

$$g(x) = \begin{cases} w & x \in \mathrm{SAT} \text{ and } w \text{ is the largest satisfying truth assignment for } x \text{ in the lexicographic order,} \\ \varepsilon & \text{otherwise.} \end{cases}$$

Here we assume that every truth assignment is a nonempty word.

Let φ be a function such that $\varphi(\varepsilon) = 0$ and for all other inputs x it holds that $\varphi(x) = 1$. For all x it holds that $f(x) = \varphi(g(x))$, and thus, $f \leq_m^p g$.

To prove that $f \not\leq_{1\text{-}T\text{-rev}}^p g$ unless $\mathrm{P} = \mathrm{NP}$, assume that $f \leq_{1\text{-}T\text{-rev}}^p g$. By definition there are three $\mathrm{FP_t}$ functions, φ, ψ, and ρ, such that for all x,

$$f(x) = \varphi(x, g(\psi(x))) \text{ and } \rho(x, f(x)) = g(\psi(x)).$$

We then construct a polynomial-time algorithm for computing f, which in turn is a polynomial-time algorithm for SAT, thereby showing that $\mathrm{P} = \mathrm{NP}$.

Let x be an input for which we want to compute the value of f and let $y = \psi(x)$. Let $w_0 = \rho(x, 0)$ and $w_1 = \rho(x, 1)$. Also, let $v_0 = \varphi(x, w_0)$ and $v_1 = \varphi(x, w_1)$. The value of $f(x)$ is either 0 or 1, so $g(y) \in \{w_0, w_1\}$, and thus, $f(x) \in \{v_0, v_1\}$. We will describe below a method for finding a $b \in \{0, 1\}$ such that $f(x) = v_b$.

Note that if $w_0 \neq \varepsilon$ and $w_0 = g(y)$, w_0 must be the largest satisfying assignment of $g(y)$ in the lexicographic order, in particular, w_0 must be a satisfying assignment of $g(y)$. So, if $w_0 \neq \varepsilon$ and w_0 is not a satisfying assignment of y, then $w_0 \neq g(y)$, so $g(y) = w_1$, and thus, $f(x) = v_1$.

Similarly, if $w_1 \neq \varepsilon$ and w_1 is not a satisfying assignment of y, then $w_1 \neq g(y)$, so $g(y) = w_0$, and thus, $f(x) = v_0$.

If neither of the above is the case, we have (either $w_0 \neq \varepsilon$ or w_0 is a satisfying assignment of $g(y)$) and (either $w_1 \neq \varepsilon$ or w_1 is a satisfying assignment of $g(y)$). If $w_0 = w_1$, then $g(y) = w_0 = w_1$, and thus $f(x) = v_0 = v_1$. If $w_0 \neq w_1$, then at least one of w_0 and w_1 is a satisfying assignment of w, which implies that y is satisfiable and so $g(y) \neq \varepsilon$. Let $b \in \{0, 1\}$ be such that w_b is the larger of w_0 and w_1. Since ε is the smallest word in the lexicographic order and no truth assignment is ε, we have $g(y) = w_b$, and thus, $f(x) = v_b$.

Since φ, ψ, and ρ are polynomial-time computable, w_0, w_1, v_0, v_1 can be computed in polynomial time. Since whether w_0 and w_1 are satisfying assignments for $g(y)$ can be decided in polynomial time, we have that f is polynomial-time computable. This proves the lemma. ■

An interesting observation regarding reversible metric reducibility is that if $f \leq^p_{\text{1-}T\text{-rev}} g$ then, in some sense, g contains a part h that is reversible metric equivalent to f via a set of two very simple reductions. Let f and g be two functions such that $f \leq^p_{\text{1-}T\text{-rev}} g$ via FP_t functions φ, ψ, and ρ. That is, $f(x) = \varphi(x, g(\psi(x)))$ and $\rho(x, f(x)) = g(\psi(x))$. We define $h(x) = g(\psi(x))$. Then we have that

$$f(x) = \varphi(x, h(x)), \tag{1}$$
$$h(x) = \rho(x, f(x)). \tag{2}$$

h is essentially the part of g that is important for the reduction from f to g. In some sense, reversible metric reducibility has a flavor of an equivalence relation.

In this paper we study many function complexity classes. It is important to know whether these classes are closed under reducibility types that we are using or not. Not surprisingly, all the classes that we study are closed under parsimonious reductions.

Proposition 2.13 *Each of* GapP*,* SpanP*,* TotP*,* #PE*, and* #P *is closed under* $\leq^p_{par}$ *reductions.*

Proof: Let $g \in \#\text{P}$ via an NP-machine M. Suppose f is $\leq^p_{par}$-reducible to g via a polynomial-time computable function ψ: for all x, $f(x) = g(\psi(x))$. Define N to be the machine that, on input x, computes $y = \psi(x)$, nondeterministically simulates M on input y, and then accepts if M accepts in the simulation and rejects otherwise. Clearly, the machine N witnesses that $f \in \#\text{P}$. The same proof works for TotP, #PE, and GapP.

Furthermore, for SpanP, consider the machine M to be a transducer and modify the algorithm of N so that it outputs the same output that M produces on each path of simulation. ■

On the other hand, even many-one reductions seem to be so powerful that the nondeterministic function classes that we study here do not seem to be closed under them. The main reason for this is that many-one reductions allow post-computation but for nondeterministic classes like #P it is difficult to get a handle on the value of their functions.

Proposition 2.14 #P *is closed under* $\leq^p_m$ *if and only if* UP = PP*.*

Proof: Ogihara and Hemaspaandra [OH93] showed that #P is closed under every polynomial-time computable operation if and only if UP = PP. Being closed under every polynomial-time operation is simply another way of saying that #P is closed under many-one reductions. ■

For similar reasons it is unlikely that either of TotP, #PE, SpanP, or GapP is closed under many-one reductions. Ogihara and Hemaspaandra [OH93] showed that unlikely complexity class collapses happen if SpanP is closed under every polynomial-time computable operation and Gupta [Gup92, Gup95] showed similar results for GapP. For the case of TotP it is easy to see that if TotP is closed under every polynomial-time computable operation then TotP = #P and then Proposition 2.14 applies. The same approach works for #PE.

We note here that each of the function classes that we discuss in this paper, FP_t, #P, #PE, TotP, SpanP, and GapP, has a canonical complete function under parsimonious reducibility. Many natural complete functions are also known.

2.4 Self-reducibility and Autoreducibility for Functions

We now define the notions of self-reducibility and autoreducibility for functions. While a large part of work in this subject has been on random-self-reducibility, we here focus on reducibility to itself achieved by deterministic computation. Because of the lack of access to randomness, the *self-reducibility* and *autoreducibility* we use here in some way mimic their language counterparts instead of random-self-reducibility.

As in the case of "many-one" type reduction, there are several ways in which one can define self-reducibility and autoreducibility for functions. One of the most natural ones is to simply mimic the definitions of those notions for languages.

Definition 2.15 *A function f is (polynomial-time) Turing autoreducible (or $\leq_T^p$ autoreducible) if there exists a polynomial-time function-oracle Turing machine M that on input x, given f as the oracle, outputs the value $f(x)$, without ever querying the oracle about x.*

A function f is (polynomial-time) length-decreasing self-reducible (or $\leq_T^p$ length-decreasing self-reducible) if f is autoreducible via a machine that queries its oracle only about words that are shorter than the machine's input.

The above definitions are the exact analogues of the definitions for the language case. Below, we define a more restrictive version, inspired by the work of Pagourtzis and Zachos [PZ06].

A *semiring* is a set S of elements with two algebraic operations $+$ and $\cdot$ with the following properties:

(i) S is closed both under $+$ and under $\cdot$; that is, for all $a, b \in S$, $a + b, a \cdot b \in S$.

(ii) S has the identity element 0 with respect to $+$ and the identity element 1 with respect to $\cdot$; that is, for all $a \in S$, $a + 0 = 0 + a = a$ and $a \cdot 1 = 1 \cdot a = a$.

(iii) S is commutative with respect to $+$; that is, for all $a, b \in S$, $a + b = b + a$.

(iv) 0 annihilates S; that is, for all $a \in S$, $0 \cdot a = a \cdot 0 = 0$.

(v) $\cdot$ satisfies the distributive law; that is, for all $a, b, c \in S$, $(a + b) \cdot c = a \cdot c + b \cdot c$ and $c \cdot (a + b) = c \cdot a + c \cdot b$.

An *encoding* of a semiring S is a one-to-one mapping from S to Σ^*. An encoding e of a semiring S is *efficient* if the following holds:

- There exists a constant c such that for all $a, b \in S$, $|e(a + b)| \leq \max\{|e(a)|, |e(b)|\} + c$.
- There exists a constant d such that for all $a, b \in S$, $|e(ab)| \leq |e(a)| + |e(b)| + d \log \max\{|e(a)|, |e(b)|\}$.
- The set $e(S)$ is in P.
- The function f_{add}, defined for all $x, y \in \Sigma^*$ by $f_{add}(x, y) = e(e^{-1}(x) + e^{-1}(y))$ if $x, y \in e(S)$ and ε otherwise, is polynomial-time computable.
- The function f_{mul}, defined for all $x, y \in \Sigma^*$ by $f_{mul}(x, y) = e(e^{-1}(x) \cdot e^{-1}(y))$ if $x, y \in e(S)$ and ε otherwise, is polynomial-time computable.

A more detailed treatment of this sort of computation by circuits can be found in the work of Borodin, Cook, and Pippenger [BCP83].

Note that each one of the set of natural numbers $\mathbb{N}$, the set of integers $\mathbb{Z}$, and the set of rationals $\mathbb{Q}$ is a semiring with an efficient encoding. Each time we use notation $\mathcal{FC}$ we mean a class of functions from Σ^* to a semiring S with an efficient encoding.

Definition 2.16 *Let f be a function from Σ^* to a semiring S with an efficient encoding. We say that f is (polynomial-time) affine autoreducible if there exists a polynomial-time computable function r with the following property: for each x, $r(x)$ is an ordered set $(t, m_1, \ldots, m_k, h_1, \ldots, h_k)$ such that $t, m_1, \ldots, m_k \in S$, $h_1, \ldots, h_k \in \Sigma^* - \{x\}$, and*

$$f(x) = t + m_1 f(h_1) + \cdots + m_k f(h_k).$$

Here, if there exists a constant ℓ such that for all x the value of k is at most ℓ, then we say that f is (polynomial-time) ℓ-affine autoreducible.

Also, if for all x, the length of each h_i appearing in $r(x)$ is smaller than the length of x ($r(\varepsilon)$ necessarily consists only of the t-part), we say that f is (polynomial-time) length-decreasing affine self-reducible.

It is easy to see that #SAT, the function that maps each propositional formula to the number of its satisfying truth assignments, is length-decreasing affine self-reducible. Let #PerfectMatching be the function that returns, given a graph as input, the number of perfect matchings of the graph. It is implicit in the paper of Pagourtzis and Zachos [PZ06] that #PerfectMatching is length-decreasing affine self-reducible, provided that the graph G is represented as a sequences of pairs (u, v) that represent its edges. In fact, they showed that this function is self-reducible for any natural representation of graphs, provided that a somewhat less restrictive notion of self-reducibility is used.

We are interested in the following two special cases of 1-affine autoreducibility.

Definition 2.17 *A function f is (polynomial-time) parsimonious autoreducible if there exists a polynomial-time computable function $h : \Sigma^* \to \Sigma^*$ such that for all x,*

$$h(x) \neq x \text{ and } f(x) = f(h(x)).$$

Definition 2.18 *Let f be a function from Σ^* to a semiring S with an efficient encoding. We say that f is (polynomial-time) nearly parsimonious autoreducible if there exist two polynomial-time computable functions $t : \Sigma^* \to S$ and $h : \Sigma^* \to \Sigma^*$ such that for all x,*

$$h(x) \neq x \text{ and } f(x) = t(x) + f(h(x)).$$

Note that if f is parsimonious autoreducible then it is nearly parsimonious autoreducible.

One can show that every polynomial-time computable function from Σ^* to $\mathbb{Z}$, the set of integers, is nearly parsimonious autoreducible. In fact, if a function f in $\mathrm{FP_t}$ from Σ^* to a semigroup S satisfies the property that there exist two distinct $a, b \in \Sigma^*$ such that

(i) for all $x \in \Sigma^* - \{a\}$, it holds that $f(x) - f(a) \in S$ and $f(x) - f(a)$ is computable in polynomial time, and

(ii) $f(a) - f(b) \in S$,

then the function h defined for all x by:

$$h(x) = \begin{cases} a & \text{if } x \neq a, \\ b & \text{otherwise,} \end{cases}$$

and the function t defined for all x by:

$$t(x) = \begin{cases} f(x) - f(a) & \text{if } x \neq a, \\ f(a) - f(b) & \text{otherwise,} \end{cases}$$

witness that f is nearly parsimonious autoreducible. As a quick corollary, we have that every polynomial-time computable function from Σ^* to $\mathbb{N}$ with two preimages of 0 is nearly parsimonious autoreducible. (Compare this result with Theorem 4.7.)

Parsimonious autoreducibility appears harder to achieve than nearly parsimonious autoreducibility because for a function f to be parsimonious autoreducible each element in $f(\Sigma^*)$, the image of f, must have at least two preimages; i.e., for every $x \in \Sigma^*$, there exists $y \in \Sigma^*$ such that $x \neq y$ and $f(x) = f(y)$. For this reason, any one-to-one function, such as the rank function, cannot be parsimonious autoreducible. On the other hand, if we consider $\mathbb{Z}$ rather than $\mathbb{N}$ to be the range of the rank function, it is nearly parsimonious autoreducible.

Lemma 2.19 *There are parsimonious complete functions for* FP_t *that are not parsimonious autoreducible.*

This lemma might offer a tool for comparing function classes against FP_t: If a class $\mathcal{FC}$ is likely to have a property that every parsimonious complete function is parsimonious autoreducible, then the class is perhaps different from FP_t. In this paper we compare the behavior of complete functions for various hard function classes, #P, SpanP, TotP, #PE and GapP, with the behavior of complete functions for FP_t. Each of #P, SpanP, TotP, #PE and GapP is, by definition, a class of functions either from Σ^* to the semiring $\mathbb{Z}$ (GapP) or to the semiring $\mathbb{N}$ (all the other classes). For such comparisons we interpret FP_t as a class of functions from Σ^* to an appropriate semiring. Detecting a different behavior of FP_t complete functions and functions complete for one of the above-mentioned classes would be a significant result showing that the power of nondeterminism is different from that of determinism.

We note here that the question of whether a function is parsimonious autoreducible is identical to the question of whether the function has a nice property that from each input x another element having the same value with respect to the function can be computed in polynomial time. That question is reminiscent of the study by Joseph and Young [JY85] of NP-complete languages in terms of polynomial-time encoding and decodable padding functions.

3 Length-Decreasing Self-Reductions

In this section we prove our results regarding length-decreasing self-reducibility and logspace length-decreasing self-reducibility.

Results of this section can be obtained using a single simple technique. For each class of our interest we construct a complete language (a complete function) with the property that the language (function) is length-decreasing self-reducible if and only if the language (function) is *easy*. Here by "easy" we mean that the language is polynomial-time decidable in the case where the self-reductions are polynomial-time computable and that the language is logarithmic-space decidable in the case where the self-reductions are logarithmic-space computable. With simple modifications, our technique applies to logspace length-decreasing self-reducibility, obtaining collapses to L.

Theorem 3.1 *Let $\mathcal{C}$ be a language class closed under polynomial-time padding with a $\leq_m^p$-complete language. If all $\leq_m^p$-complete languages for $\mathcal{C}$ are length-decreasing self-reducible then $\mathcal{C} \subseteq \mathrm{P}$.*

Proof: Let A be an arbitrary $\mathcal{C}$-complete language under polynomial-time many-one reductions. Let $k \geq 2$ be an integer. Let f be a function from $\mathbb{N}$ to $\mathbb{N}$ defined for all $n \geq 0$ by: $f(n) =$ the smallest integer 2^{k^i} such that i is an integer and $2^{k^i} > n$. Define B as follows:

$$B = \{x10^m \mid x \in A \wedge |x| + 1 + m = f(|x|)\}.$$

Note that B is simply a padded version of A, in which the length of each member of A is inflated to an integer of the form 2^{k^i}. The function f is clearly a pad-length function. Also, f is polynomially bounded because for all $n \geq 0$ it holds that $f(n) \leq 2 + n^k$. Furthermore, f is logspace computable: On input x, $f(x)$ can be computed by successively calculating in binary $2, 2^k, (2^k)^k, ((2^k)^k)^k, \cdots$ until the value exceeds $|x|$. Thus, by Lemma 2.6, B is $\mathcal{C}$-complete. Then by our assumption B is length-decreasing self-reducible.

Since B is length-decreasing self-reducible, there is a deterministic Turing machine M such that:

1. M runs in polynomial time;
2. $L(M^B) = B$;
3. On input x, M queries its oracle only about words shorter than x.

We now describe a polynomial-time algorithm for B that does not need the oracle. On input x our algorithm behaves as follows: If $|x|$ is not of the form 2^{k^i}, then by definition x is clearly a non-member of B, so we immediately terminate the simulation (or return from the simulation if we are dealing with a recursive call) by asserting that $x \notin B$. Otherwise, we simulate M on input x replacing each oracle query by a recursive call to M. Nontrivial simulations of M take place only when the length of the input is of the form 2^{k^i} for some integer i. Since M is length-decreasing self-reducible, we can assume that on inputs of appropriate length all oracle queries regard words of length at most $|x|^{\frac{1}{k}}$. We also modify M to use a look-up table to decide words of length at most 2.

Now we are ready to prove that our algorithm runs in polynomial time. Let p be a polynomial such that for all x the running time of M on input x, assuming that the cost of each oracle query is 1, is $p(|x|)$. Note that all polynomials we are concerned with have nonnegative coefficients, so p is strictly increasing. We also assume that for all nonnegative n it holds that $p(n) > 1$. What is the time complexity of our algorithm? In the recursion tree, there are at most $\lceil \log \log n \rceil$ levels because words of length at most $n^{\frac{1}{k^{\lceil \log \log n \rceil}}}$ are of length at most 2, and we have a look-up table to decide whether we accept them or not. For each level $i \geq 0$ of the recursion tree, let P_i be the number of computational steps other than processing of recursive calls required to simulate all the level-i simulations. It holds that:

$$\begin{aligned} P_0 &\leq p(n), \\ P_1 &\leq p(n) \cdot p(n^{\frac{1}{k}}), \\ P_2 &\leq p(n) \cdot p(n^{\frac{1}{k}}) \cdot p(n^{\frac{1}{k^2}}), \end{aligned}$$

and so on. This is because at the 0^{th} level we just have one call to M with input of length n. At this level M can make at most $p(n)$ oracle queries, each of length at most $n^{\frac{1}{k}}$, thus at the first level we need to execute at most $p(n) \cdot p(n^{\frac{1}{k}})$ basic steps. The same analysis applies to P_2, P_3, and so on, up to $P_{\lceil \log \log n \rceil}$.

Let us now estimate the value of P_j for some arbitrary j. We can assume without loss of generality that $p(n) \leq n^d$ for some nonnegative integer d. It holds that:

$$P_j \leq \prod_{i=0}^{j} p(n^{\frac{1}{k^i}}) \leq \prod_{i=0}^{j} n^{\frac{d}{k^i}} = n^{\sum_{i=0}^{j} \frac{d}{k^i}} \leq n^{\sum_{i=0}^{\infty} \frac{d}{k^i}} = n^{\frac{dk}{k-1}}.$$

Note that the final result does not depend on j so the time complexity of the whole algorithm is bounded by:

$$\lceil \log \log n \rceil \cdot n^{\frac{dk}{k-1}} \in O(n^h),$$

where h is some nonnegative integer such that $h \geq \frac{dk}{k-1} + 1$. Thus, we have shown that B is decidable in polynomial time. Since B is complete for $\mathcal{C}$, it holds that all languages in $\mathcal{C}$ are decidable in polynomial time. This proves the theorem. ■

For all classes $\mathcal{C}$ other than some pathological cases ($\mathcal{C} = \{\emptyset\}$ or $\mathcal{C} = \{\Sigma^\star\}$), we have, by the above theorem, that if $\mathcal{C}$ is closed under many-one reductions and all $\mathcal{C}$-complete sets are length-decreasing self-reducible (and $\mathcal{C}$ has at least one complete set) then $\mathcal{C} \subseteq$ P. Thus, we have the following corollaries:

Corollary 3.2 *If all* NP*-complete languages are length-decreasing self-reducible, then* P = NP*.*

Corollary 3.3 *If all* PSPACE*-complete languages are length-decreasing self-reducible then* P = PSPACE*.*

A similar theorem holds for length-decreasing self-reducibility in logarithmic space.

Theorem 3.4 *Let $\mathcal{C}$ be a language class that is closed under logspace padding and has a $\leq_m^{\log}$-complete language. If all $\leq_m^{\log}$-complete languages for $\mathcal{C}$ are logspace length-decreasing self-reducible then $\mathcal{C} \subseteq$ L.*

Proof: The proof is essentially the same as the proof of Theorem 3.1. The only difference is that now we need to make sure that we do not use more than a logarithmic amount of space for our recursive calls.

Let A, k, f, B, and M be as defined in the proof of Theorem 3.1 with the exception that A is logspace many-one complete for $\mathcal{C}$, and that M is an oracle Turing Machine witnessing that A is logspace length-decreasing self-reducible. We need to implement the following space-preserving strategy. We cannot generate the input for the recursive calls on the tape because that would use more than a logarithmic amount of space. Instead, after each recursive call that completed, we store the contents of the tape and the position of the head (replacing the previously stored one) and run the machine—without writing out the next query word—until finally a subsequent query is to be asked (recursive call is to be performed). Then, in that recursive call we only pass the stored tape so that the recursive call can recreate any of the bits of its input word on demand (using an at most $\log n$ bit counter).

By definition of logspace length-decreasing self-reducibility, excluding the recursive calls, our machine uses at most logarithmic amount of space. Thus, if the input word has length n then for the recursive calls at the first level we only need at most $c \log n$ bits, where c is some constant. Consequently, the total amount of space that we need to handle each branch of recursion is:

$$\sum_{i=0}^{\lceil \log \log n \rceil} c \log \left(n^{\frac{1}{k^i}} \right) \quad \leq \quad \sum_{i=0}^{\infty} c \log \left(n^{\frac{1}{k^i}} \right)$$

$$
\begin{aligned}
&= c \log n^{\sum_{i=1}^{\infty} \frac{1}{k^i}} \\
&= c \left(\frac{k}{k-1} \right) \log n.
\end{aligned}
$$

Since we handle recursion branches one at a time, this means that our algorithm requires at most logarithmic space. As B is $\mathcal{C}$-complete (with respect to logspace many-one reductions) it holds that all languages in $\mathcal{C}$ can be decided in logarithmic space. ■

The above theorem yields the following corollary.

Corollary 3.5 *If all $\leq_m^{\log}$-complete sets for* NL *are logspace length-decreasing self-reducible then* L = NL.

By the Time and Space Hierarchy Theorems, it holds that

- L $\neq$ PSPACE and
- P $\neq$ EXP.

Now by Theorems 3.1 and 3.4, it holds that there is a PSPACE-complete language that is not logspace length-decreasing self-reducible, and that there exists an EXP-complete language that is not length-decreasing self-reducible.

We note that it follows from Theorem 3.1 that if P $\neq$ PSPACE then there are PSPACE-complete sets that are not length-decreasing self-reducible. However, it was proved by Beigel and Feigenbaum [BF92] (see also the work of Glaßer *et al.* [GOP$^+$05]) that all PSPACE-complete languages are autoreducible. Thus, if P $\neq$ PSPACE then the notions of polynomial-time length-decreasing self-reducibility and polynomial-time autoreducibility are different.

Let us now turn to the issue of length-decreasing self-reductions of complete functions. Not surprisingly, the answer, and the technique to obtain it, is the same as in the case of languages. The following theorem is the exact counterpart of Theorem 3.1 for the case of functions.

Theorem 3.6 *For every function $f : \Sigma^* \to \Sigma^*$, there exists a function $f' : \Sigma^* \to \Sigma^*$ such that*

- *$f(\Sigma^*) = f'(\Sigma^*)$, $f \leq_{par}^p f'$, $f' \leq_{par}^p f$, and*
- *if f' is Turing length-decreasing self-reducible then both f' and f are polynomial-time computable.*

Proof: Let f be an arbitrary function from Σ^* to Σ^*. Let k be any integer greater than 1. Let $S = \{2^{k^n} \mid n \in \mathbb{N}\}$. For each positive integer m and for each word x in $\Sigma^m - \{1^m\}$, let $\rho_m(x)$ be the word y of length at most $m-1$ whose rank in Σ^* is equal to the rank of x in Σ^m. For example, $\rho_3(000) = \varepsilon$, $\rho_3(011) = 00$, and $\rho_3(110) = 11$. For all x define $f'(x)$ by:

$$
f'(x) = \begin{cases} f(\rho_{|x|}(x)) & \text{if } |x| \in S \text{ and } x \notin 1^*, \\ f(\varepsilon) & \text{otherwise.} \end{cases}
$$

It is easy to see that f is parsimonious reducible to f' by the function that maps each x to $\rho_m^{-1}(x)$, where m is the smallest integer in S that is greater than $|x|$. This reduction is polynomial-time computable, since for each x the value m does not exceed $|x|^k$.

It is also easy to see that f' is parsimonious reducible to f by the function that maps each x to $\rho_{|x|}(x)$ if $|x| \in S$ and $x \notin 1^*$ and to ε otherwise.

Now suppose that f' is length-decreasing self-reducible via a function-oracle Turing machine M. We can assume that M never queries its oracle about a word whose length is not in S or about a word over $\{1\}$ because for these words the value is known to be $f(\varepsilon)$, which can be treated as a constant. Then, on input x, we can compute $f'(x)$ recursively using the following algorithm:

- If either $|x| \notin S$ or $x \in 1^*$, then output $f(\varepsilon)$.
- Otherwise, simulate M on input x. During this simulation, whenever M makes a query, say y, to its oracle, simulate M on y using a recursive call to obtain the answer of the oracle.

Using exactly the same argument as in the proof of Theorem 3.1, we can show that this algorithm correctly computes $f'(x)$ in polynomial time.

Since $f \leq^p_{par} f'$, if f' is computable in polynomial time, so is f. This proves the theorem. ■

Armed with Theorem 3.6 we can show that, for example, not all $\leq^p_{par}$-complete functions for #P are length-decreasing self-reducible unless $\#\mathrm{P} \subseteq \mathrm{FP_t}$. The same can be shown for many other function classes.

Corollary 3.7 *Let $\mathcal{FC}$ be one of* GapP*,* SpanP*,* TotP*,* #PE*, and* #P*. If all $\leq^p_{par}$-complete functions for $\mathcal{FC}$ are length-decreasing self-reducible, then $\mathcal{FC} \subseteq \mathrm{FP_t}$.*

Proof: Let $\mathcal{FC}$ be as in the hypothesis of the corollary. Then $\mathcal{FC}$ has a $\leq^p_{par}$-complete function f. Let f' be the function whose existence is given in Theorem 3.6. Then, $f \leq^p_{par} f'$, $f' \leq^p_{par} f$, $f(\Sigma^*) = f'(\Sigma)$, and furthermore, if f' is length-decreasing self-reducible then f' is polynomial-time computable. By Proposition 2.13, $\mathcal{FC}$ is closed under $\leq^p_{par}$ reductions, so $f' \in \mathcal{FC}$. Since $f \leq^p_{par} f'$, this implies that f' is $\leq^p_{par}$-complete for $\mathcal{FC}$. Now, assume that all $\leq^p_{par}$-complete functions for $\mathcal{FC}$ are length-decreasing self-reducible. Then, we have that f' is length-decreasing self-reducible. This implies $f' \in \mathrm{FP_t}$. Since f' is $\leq^p_{par}$-complete for $\mathcal{FC}$, we have $\mathcal{FC} \subseteq \mathrm{FP_t}$. ■

Since parsimonious reducibility is the least flexible reducibility type for functions Corollary 3.7 is in some sense in the strongest possible form for the case of length-decreasing self-reducibility.

4 Autoreductions of Complete Functions

We now turn to the issue of autoreducibility of complete functions. In the previous section we showed that for many natural function complexity classes it is not the case that all their parsimonious complete functions are length-decreasing self-reducible. In contrast, we show in this section that for many classes all their complete functions are indeed autoreducible and that in many cases the parsimonious-complete functions are in fact nearly parsimonious autoreducible. We then make observations about algebraic properties of parsimonious complete functions.

We say that a class of functions $\mathcal{FC}$ from Σ^* to $\mathbb{N}$ ($\mathbb{Z}$) is *closed under increment* (respectively, *closed under proper decrement*) if for all $f \in \mathcal{FC}$, the function f' defined for all x by $f'(x) = f(x)+1$ (respectively, $f'(x) = \max\{0, f(x) - 1\}$) belongs to $\mathcal{FC}$. It is not hard to see that GapP, SpanP, TotP, #PE, and #P are closed under increment.

Proposition 4.1 *Each one of* GapP*,* SpanP*,* TotP*,* #PE*, and* #P *is closed under increment. Also,* TotP *is closed under proper decrement.*

It is not known whether the other three classes are closed under proper decrement (see [OH93]).

Theorem 4.2 shows that if a class of functions is closed under increment and has a $\leq^p_{par}$-complete function all of its $\leq^p_{par}$-complete functions are autoreducible with one query.

Theorem 4.2 *Let $\mathcal{FC}$ be a class of functions that is closed under increment and has parsimonious complete functions. Then every $\leq^p_{par}$-complete function for $\mathcal{FC}$ is autoreducible with one query.*

Proof: Let $\mathcal{FC}$ be as in the hypothesis of the theorem and let f be a $\leq^p_{par}$-complete function for $\mathcal{FC}$. We will construct a polynomial-time function-oracle Turing machine M witnessing that f is autoreducible with one query.

Define $g(x) = f(x) + 1$. Since $\mathcal{FC}$ is closed under increment, $g \in \mathcal{FC}$. Then there is a $\leq^p_{par}$-reduction ψ from g to f; that is, for all x it holds that $g(x) = f(\psi(x))$. Let M be a machine that on input x behaves as follows:

1. M computes $y = \psi(x)$.
2. M queries the oracle about y and obtains $z = f(y)$ as the answer.
3. M outputs $z - 1$.

For all x, $f(\psi(x)) = g(x) = f(x) - 1$ so M correctly computes $f(x)$. For no x it holds that $\psi(x) = x$; otherwise, we would have $f(x) = f(x) - 1$ for some x, a contradiction. This proves the theorem. ■

Corollary 4.3 *Each class $\mathcal{FC}$ chosen from* GapP*,* SpanP*,* TotP*,* #PE*, and* #P *has the property that all of its $\leq^p_{par}$-complete functions are autoreducible with one query.*

The above results raise the question of whether a similar property holds for reducibility notions that are more flexible than parsimonious reducibility and for more demanding versions of autoreducibility. That is, under which condition every complete function for a class $\mathcal{FC}$ possesses a certain type of autoreducibility.

Beigel and Feigenbaum showed that if a language class $\mathcal{C}$ has a length-decreasing self-reducible language then every single Turing-complete language for $\mathcal{C}$ is autoreducible. We prove several results of a similar flavor for function classes.

Theorem 4.4 *Let $\mathcal{FC}$ be a function class that has $\leq^p_{par}$-complete functions. If $\mathcal{FC}$ has a $\leq^p_{par}$-complete function that is length-decreasing affine self-reducible, then every $\leq^p_{par}$-complete function for $\mathcal{FC}$ is affine autoreducible.*

Proof: Let $\mathcal{FC}$ be a function class having a $\leq^p_{par}$-complete function. Let f be a $\leq^p_{par}$-complete function for $\mathcal{FC}$. Let S be a semiring with an efficient encoding associated with the class $\mathcal{FC}$. Suppose that f is length-decreasing affine self-reducible. Then there exists a polynomial-time computable function r with the following property: For all x, $r(x)$ is an ordered set $(t, m_1, \ldots, m_k, h_1, \ldots, h_k)$ such that $t, m_1, \ldots, m_k$ are members of S, $h_1, \ldots, h_k$ are words having lengths less than $|x|$, and

$$f(x) = t + m_1 f(h_1) + \cdots + m_k f(h_k).$$

Let g be an arbitrary $\leq^p_{par}$-complete function for $\mathcal{FC}$. Let ψ be a $\leq^p_{par}$-reduction from f to g and let φ be a $\leq^p_{par}$-reduction from g to f. We will show that g is affine autoreducible.

Consider the machine M that on input x sets $y = \varphi(x)$ and then executes the following algorithm:

Step 1 Compute $r(y) = (t, m_1, \ldots . m_k, h_1, \ldots, h_k)$.

Step 2 If $k = 0$, then output (t) and halt.

Step 3 ($k \geq 1$.) For each i, $1 \leq i \leq k$, compute $q_i = \psi(h_i)$. If for some i, $1 \leq i \leq k$, $q_i = x$, pick the smallest such i, set y to h_i, and then return to Step 1.

Step 4 ($k \geq 1$ and for all i, $1 \leq i \leq k$, $q_i \neq x$.) Output $(t, m_1, \ldots, m_k, q_1, \ldots, q_k)$ and halt.

The initial value of y is $\varphi(x)$. Since φ is a $\leq^p_{par}$-reduction from g to f, it holds that $g(x) = f(y) = f(\varphi(x))$. If y is replaced by some h_i in Step 3, then $\psi(h_i) = x$. Since ψ is a $\leq^p_{par}$-reduction from g to f, it holds that $g(x) = f(h_i)$. Thus, during the execution of the algorithm, the property $g(x) = f(y)$ is maintained.

If M outputs (t) in Step 2, then we have $f(y) = t$, and thus, it holds that $g(x) = t$. If M outputs $(t, m_1, \ldots, m_k, q_1, \ldots, q_k)$ in Step 4, then we have $f(y) = t + m_1 f(h_1) + \cdots + m_k f(h_k)$ and for all i, $1 \leq i \leq k$, we have $\psi(h_i) = q_i$. Since ψ is a $\leq^p_{par}$-reduction from f to g, for all i, $1 \leq i \leq k$, it holds that $g(q_i) = f(h_i)$. Since $f(x) = g(y)$ is a loop-invariant, we thus have

$$g(y) = t + m_1 g(q_1) + \cdots + m_k g(q_k).$$

Thus, M correctly computes $g(x)$ with g as the oracle without having to query x.

It now suffices to show that M runs in polynomial time. If some h_i replaces y in Step 3, then due to the length-decreasing nature of r, $|h_i| < |y|$. So, each execution of the loop can be done in time polynomial in $|y|$, and thus, in time polynomial in $|x|$. Also, the above condition on the length of h_i's implies that the loop is executed at most $|\varphi(x)|+1$ times. Thus, M runs in polynomial time. ■

One wonders whether a result similar to Theorem 4.4 holds for $\leq^p_T$-complete functions. We don't have an absolute answer to the question, but offer a partial answer, which is stated below.

Theorem 4.5 *Let $\mathcal{FC}$ be a function class that has both Turing-complete and reversible metric complete functions. If a $\leq^p_T$-complete function for $\mathcal{FC}$ is length-decreasing self-reducible, then each reversible metric complete function for $\mathcal{FC}$ is Turing autoreducible.*

Proof: Suppose that the hypothesis of the theorem holds. Let f be a $\leq^p_T$-complete function for $\mathcal{FC}$ that is length-decreasing self-reducible and let g be an arbitrary $\leq^p_{1\text{-}T\text{-rev}}$-complete function for $\mathcal{FC}$. Let R be a polynomial-time function-oracle Turing machine that computes a $\leq^p_T$-reduction from g to f and let S be a machine witnessing that g is length-decreasing self-reducible. Since g is $\leq^p_{1\text{-}T\text{-rev}}$-complete, there exist some $\varphi, \psi, \rho \in \mathrm{FP_t}$ witnessing that $f \leq^p_{1\text{-}T\text{-rev}} g$; that is, for all y, it holds that

$$f(y) = \varphi(y, g(\psi(y))) \text{ and } g(\psi(y)) = \rho(y, f(y)). \tag{3}$$

It suffices to show that g is $\leq^p_T$ autoreducible. Consider a machine N that on input x executes the following algorithm:

Phase 1 Attempt to compute $g(x)$ by simulating R on input x. For each query y of R, execute the following:

(a) Compute $x' = \psi(y)$.

(b) If $x' = x$, then terminate the simulation, set y' to y, and then jump to Phase 2.

(c) If $x' \neq x$, then ask the oracle about x' and obtain its answer a.

(d) Compute $b = \varphi(y, a)$. Return to the simulation of R with b as the answer.

When the simulation of R is completed with an output c, output c and halt.

Phase 2 Attempt to simulate S on input y' by answering each query y exactly in the same way as in Phase 1. In particular, upon finding a query y such that $\psi(y) = x$ in Step (b), restart Phase 2 with the value of y' set to that query y.

When the simulation of S is completed with an output d, output $\rho(y', d)$ and halt.

It is clear that N on x never asks its oracle about x. We show that for all x, the output of N on x is $g(x)$. Let x be fixed. Suppose that N on input x outputs c in Phase 1 with f as the oracle. Then this c is the output of R on input x provided that each query y of R is answered by $\varphi(y, g(\varphi(y))$. Then, by (3) in the above, the answer to query y is equal to $f(y)$. This means that c is precisely the output of R on input x with oracle f, which is $g(x)$. This also implies that, if N does not enter Phase 2, then N halts and outputs $g(x)$. So, suppose that N on input x enters Phase 2. Whenever N enters Phase 2, it holds that $\psi(y') = x$, and thus, by (3) in the above, it holds that $g(x) = \rho(y', f(y'))$. As in the case when N halts in Phase 1, if N finishes simulation of S with an output value of d, this d is equal to $f(y')$. Thus, $\rho(y', d)$, which is the output of N in Phase 2, is equal to $g(x)$.

Let p be a polynomial that bounds both the running time of R with oracle g and the running time of S with oracle g. If the value of y' is changed from y_1 to y_2, we have that y_2 is a query word of S on input y_1 with g as the oracle. This means that $|y_2| < |y_1|$. If y_0 is the value of y' when N enters Phase 2 for the first time, then y' is updated at most $|y_0| + 1$ times, and this is at most $p(|x|) + 1$. This implies that Phase 2 eventually halts with the correct value of $g(x)$.

The time that it takes to simulate S and R, except for the time to compute φ, ρ, and ψ is $O(p(|x|)^2)$. Since each query y produced in the simulation has length bounded by $p(|x|)$ and φ, ρ, and ψ are polynomial-time computable, the total running time of the above algorithm is bounded by a fixed polynomial in $|x|$. This proves the theorem. ■

A crucial part of the proofs of Theorems 4.4 and 4.5 is the use of *reversible* metric reducibility. (Recall that parsimonious reducibility is a special type of a reversible metric reducibility.) The next theorem shows that a result similar to Theorems 4.4 and 4.5 holds without assuming reversibility.

Theorem 4.6 *Let $\mathcal{FC}$ be a function class that has both $\leq^p_T$-complete and $\leq^p_m$-complete functions. If there is a $\leq^p_T$-complete function for $\mathcal{FC}$ that is length-decreasing $\leq^p_T$ self-reducible, then each $\leq^p_m$-complete function for $\mathcal{FC}$ is $\leq^p_T$ autoreducible.*

Proof: Let $\mathcal{FC}$ be as in the hypothesis of the theorem. Let g be an arbitrary $\leq^p_m$-complete function for $\mathcal{FC}$. Let f be an arbitrary $\leq^p_T$-complete function for $\mathcal{FC}$ that is length-decreasing self-reducible. Let S be a machine witnessing that f is length-decreasing self-reducible. Let R be a machine that $\leq^p_T$-reduces g to f. Since g is $\leq^p_m$-complete for $\mathcal{FC}$, there exist two $\mathrm{FP_t}$ functions φ and ψ such that for all y, $f(y) = \varphi(g(\psi(y)))$. It then suffices to show that g is $\leq^p_T$-autoreducible.

Let x be fixed for which we wish to compute the value of g. The idea behind the algorithm below is that we compute the Turing reduction from g to f and we use the fact that $f \leq^p_m g$ to answer each query y regarding f using the fact that $f(y) = \varphi(g(\psi(y)))$. However, we cannot do that in case $\psi(y) = x$. Yet, we notice that for every two strings y_1 and y_2 such that $\psi(y_1) = \psi(y_2)$ it holds that $f(y_1) = f(y_2)$. We use this fact to compute, using the self-reducibility of f, $K = f(y)$ in the case when $\psi(y) = x$. Naturally, we only need to compute K once and reuse it each time a query y with $\psi(y) = x$ is posed.

Step 1 Set $F = 0$. Start simulation of R on input x. For each query y of R, do the following:

(a) compute $z = \psi(y)$;

(b) if $z \neq x$, query z to the oracle, obtain an answer a, and return to the simulation of R with $\varphi(a)$ as the answer;

(c) if $z = x$ and $F = 1$, then return to the simulation of R with K as the answer;

(d) if $z = x$ and $F = 0$, suspend the simulation of R, set w to y, execute Step 2 and then return to the simulation of R with K as the answer.

Step 2 Simulate S on input w. For each query u of R, do the following:

(a) compute $v = \psi(u)$,

(b) if $v \neq x$, query v to the oracle, obtain an answer b, and return to the simulation of S with $\varphi(b)$ as the answer;

(c) if $z = x$, then set w to u and start over Step 2.

If the simulation of S is completed, set $F = 1$, set K to the output of S, and return to the suspended simulation of R.

Step 3 When the simulation of R is completed with an output c, output c and halt.

Excluding the cost of computing K, our algorithm clearly runs in polynomial time. Also, it should be clear that if the value K is computed correctly then our algorithm returns the correct value.

We will now argue that executing Step 2 takes at most polynomial time and computes the correct value of K. Let ℓ be the length of the longest query of R on input x. While computing K in Step 2, each time w is replaced by a new element and the simulation of S is started over, w becomes shorter due to the length-decreasing nature of S. Thus, the number of times that the replacement of w occurs is bounded by ℓ, and thus, the number of times that S is simulated as well as the length of the longest input to S is bounded by ℓ. Since S is polynomial-time bounded, Step 2 takes polynomial time.

Step 2 computes the correct value of K: If it completes without ever starting over (e.g., without reaching a query u such that $\psi(u) = x$) then, by definition of S, K is set to $f(w)$. On the other hand, each time we reach Step 2(c) it holds that $f(w) = f(u)$ and thus we can safely restart S with input u.

From the above discussion, our algorithm witnesses that g is autoreducible. This proves the theorem. ■

#SAT is a length-decreasing parsimonious-complete function for #P and so, by Theorem 4.4, we have that all #P parsimonious-complete functions are affine autoreducible. Similarly, all many-one complete or reversibly metric complete functions for #P are Turing autoreducible (via Theorems 4.6 and 4.5). An analogous result holds for random-self-reducibility. Feigenbaum and Fortnow [FF91] showed that all #P Turing-complete functions are adaptively random-self-reducible with polynomially many queries.

In the study of reducibility and autoreducibility, the number of queries that a machine that computes the reducibility/autoreducibility must make is an important subject of investigation. In the case of random-self-reducibility, the results of Abadi, Feigenbaum, and Kilian [AFK89] and of Feigenbaum, Kannan, and Nisan [FKN90] show that the number of queries cannot be very small. In particular, #P parsimonious-complete functions are not random-self-reducible with one query unless the polynomial hierarchy collapses to its third level.[5] Interestingly, in the case of deterministic autoreductions we can show that a single query is sufficient.

[5]Note that there are slight differences in definitions of random-self-reductions between [AFK89] and [FKN90]

Theorem 4.7 *Each parsimonious complete function for* #P *that has two preimages of* 0 *is nearly parsimonious autoreducible.*

Proof: Let f be an arbitrary parsimonious-complete function for #P such that $f(x) = 0$ holds for at least two distinct values of x and let x_1 and x_2 be two such inputs. Let M be an NP machine such that $f = \#\mathrm{acc}_M$. Without loss of generality, we may assume that (a) each nondeterministic choice of M is binary, and (b) there exists a polynomial p with a positive constant term such that for each input x, the computation tree of M on input x is a complete binary tree of depth $p(|x|)$; that is, each computation path of M on x uses exactly $p(|x|)$ nondeterministic moves. Then, for each input x, each computation path of M on input x can be described as a binary word of length $p(|x|)$. We also assume that for every word x, the rightmost computation path, the path $1^{p(|x|)}$, is rejecting.

For each x and y, $|y| = p(|x|)$, let $g(x, y)$ be the number of accepting paths π of M on input x such that $\pi < y$. Clearly, for all x, $g(x, 1^{p(|x|)}) = f(x)$ and the function g is in #P. By our assumption, g is then parsimonious reducible to f via some $\mathrm{FP_t}$ function ψ.

We will construct a polynomial-time computable function r such that for all x, $r(x) = (t, y)$, $t \in \mathbb{N}$, $y \in \Sigma^*$, and $f(x) = t + f(y)$. The function r is computed by way of the following algorithm, inspired by the one presented in [GOP+05] to prove that every NP-complete set is autoreducible. On input x, do the following:

Step 1 Compute $x' = \psi(x, 1^{p(|x|)})$.

Step 2 If $x' \neq x$ then output $(0, x')$.

Step 3 ($x' = x$.) Compute $x'' = \psi(x, 0^{p(|x|)})$.

Step 4 If $x'' = x$ then output $(0, x_1)$ if $x_1 \neq x$ and $(0, x_2)$ otherwise.

Step 5 ($x'' \neq x$) , then it holds that $\psi(x, 0^{p(|x|)}) \neq \psi(x, 1^{p(|x|)}) = x$, so execute binary-search to find a word y of length $p(|x|)$ such that $\psi(x, y) \neq x$ and $\psi(x, \mathrm{succ}(y)) = x$. Output $(0, \psi(x, y))$ if y is a rejecting path of M on input x and $(1, \psi(x, y))$ otherwise.

Clearly, this algorithm runs in polynomial time.

Suppose that the algorithm outputs $(0, x')$ in Step 2. We then have that $x \neq x'$ and $f(x') = g(x, 1^{p(|x|)}) = \#\mathrm{acc}_M(x)$, so we have that $f(x) = 0 + f(x')$, and thus, the output of the algorithm is correct. Suppose that the algorithm outputs $(0, u)$ in Step 4. Since $x = x'' = \psi(x, 0^{p(|x|)})$ and $g(x, 0^{p(|x|)}) = 0$, it must be the case that $f(x) = 0$. Since $f(x_1) = f(x_2) = 0$, and u is chosen from x_1 and x_2 so that $u \neq x$ holds, we have that $f(x) = 0 + g(y)$, and thus, the output of the algorithm is correct. Finally, suppose that the algorithm outputs (b, u) for some $b \in \{0, 1\}$ and some u. The value of u is chosen so that $\psi(x, u) \neq x$ and $\psi(x, \mathrm{succ}(u)) = x = \psi(x, 1^{p(|x|)})$. Since $\psi(x, 1^{p(|x|)}) = f(x)$, we have that $f(x) = f(u)$ if u is a rejecting path of M on x and $f(x) = f(u) + 1$ if u is an accepting path of M on x. So, the output of the algorithm is correct in either case. Thus, the algorithm correctly computes a nearly parsimonious autoreduction of f. ■

The above proof raises a couple of interesting points, which we will address in the remainder of this section.

First, let us observe that an analogous result can be obtained for many-one complete #P functions.

Corollary 4.8 *For every many-one complete* #P *function* f *that takes the value* 0 *for at least two words,* x_1 *and* x_2*, there exist three* $\mathrm{FP_t}$ *functions* t*,* h*, and* h' *such that for all words* x

$$f(x) = t(x) + h'(f(h(x))),$$

and $h(x) \neq x$*.*

Proof: The proof is identical to the proof for the case of parsimonious-complete #P functions, except that we need to take into account the fact that instead of having a parsimonious reduction function ψ, we have two functions ψ_1 and ψ_2, such that for all x it holds that

$$f(x) = \psi_2(g(\psi_1(x))),$$

where g is the function defined in the proof of Theorem 4.7. It is then sufficient to replace each occurrence of ψ in the proof of Theorem 4.7 with ψ_1, and then apply $h' = \psi_2$ to $f(h(x))$. ■

We note here that both Theorem 4.7 and Corollary 4.8 also hold for TotP.

Another interesting issue is the nature of the function $t(x)$ in the affine autoreductions of Theorem 4.7. The additive term t that appears in the value of r is either 0 or 1, but it seems hard to predict in polynomial time for which x the value of t is 1. Is it possible to give, for each #P parsimonious-complete function, an autoreduction where $t(x)$ behaves in a more stable and predictable fashion? In particular, can we get $t(x) = 0$ for all x? If we were able to do so, we would have separated $\mathrm{FP_t}$ and #P: All #P parsimonious-complete functions would be parsimonious autoreducible and we know, by Lemma 2.19, that not all $\mathrm{FP_t}$ parsimonious-complete functions are parsimonious autoreducible.

On the other hand, perhaps we can make $t(x) = 1$ for all words x such that $f(x) > 0$ and $t(x) = 0$ otherwise? This, however, seems even more unlikely. Such a function t would have the property that $t(x) = 1$ if $f(x) > 0$ and $t(x) = 0$ otherwise. Thus, one could use it to decide in polynomial time the membership in the language $L = \{x \mid f(x) > 0\}$, which is equal to SAT with $f = \#\mathrm{SAT}$. We show, however, that such autoreductions are possible for parsimonious complete functions for TotP.

Corollary 4.9 *Every* TotP *parsimonious complete function* f *that takes value* 0 *for at least two arguments,* x_1 *and* x_2*, is nearly parsimonious autoreducible via two* $\mathrm{FP_t}$ *functions,* t *and* h*, such that for each* x*:*

$$f(x) = t(x) + f(h(x))$$

and (1) $h(x) \neq x$*, (2) if* $f(x) > 0$ *then* $t(x) = 1$*, and (3)* $t(x) = 0$ *otherwise.*

Proof: TotP is closed under proper decrement and for each TotP function f it is possible to test in polynomial time whether $f(x) > 0$.

Let f be a parsimonious-complete TotP function. We define the functions t and h that constitute f's autoreduction with the properties promised by the theorem. Let $t(x) = 1$ if $f(x) > 0$ and let $t(x) = 0$ otherwise. Clearly, t belongs to $\mathrm{FP_t}$.

To define h, we need a helper function g, $g(x) = f(x) - t(x)$. Since TotP is closed under proper decrement we have $g \in \mathrm{TotP}$. Let ψ be an $\mathrm{FP_t}$ function via which g parsimoniously reduces to f. We define $h(x)$ as follows.

$$h(x) = \begin{cases} \psi(x) & \text{if } f(x) > 0, \\ x_1 & \text{if } x = x_2, \\ x_2 & \text{otherwise.} \end{cases}$$

Clearly, h is computable in polynomial time. Let us show that $h(x) \neq x$ and that $f(x) = t(x) + f(h(x))$ for each x. If $f(x) > 0$ then $f(\psi(x)) = g(x) = f(x) - 1$ and so naturally $h(x) = \psi(x) \neq x$. Also, then $f(x) = f(\psi(x)) + 1 = f(h(x)) + t(x)$. If $f(x) = 0$ then $t(x) = 0$, $f(h(x)) = 0$, and naturally $h(x) \neq x$. This proves the corollary. ■

Let us come back to the discussion of the autoreducibility of #P parsimonious-complete functions. For these functions how close we can get to having parsimonious autoreductions? By Theorem 4.2, for each such function f and input x we can produce $x' \neq x$ such that $f(x') = f(x) + 1$. On the other hand, by the proof of Theorem 4.7, we can produce $x'' \neq x$ such that $f(x'')$ equals either $f(x)$ or $f(x) - 1$. Yet, producing $x''' \neq x$ such that $f(x''') = f(x)$ seems to be very difficult: The ability to do so would separate #P from FP_{t}.

A separate issue regarding Theorem 4.7 is the requirement that each parsimonious-complete function f that it operates on needs to assume value 0 for at least two words, x_1 and x_2. This requirement is very natural. Every parsimonious-complete function f has to assume value 0 on at least one input because it must be possible to parsimoniously reduce the function $z(x) = 0$ to it. Let x_0 be some word such that $f(x_0) = 0$. If there was no other word x such that $f(x) = 0$ then we would have $f \in \#\mathrm{PE}$ as to test if $f(x) = 0$ it would suffice to check if $x = x_0$. Since $\#\mathrm{PE} = \#\mathrm{P}$ if and only if $\mathrm{P} = \mathrm{NP}$, we have the following corollary.

Corollary 4.10 *There exists a parsimonious-complete* #P *function f such that f takes the value* 0 *for exactly one argument, x_0, if and only if* $\mathrm{P} = \mathrm{NP}$.

Pushing this idea even further, let us ask ourselves a slightly different question: What consequences follow if there is a parsimonious complete function for #P that is one-to-one?

Theorem 4.11 *Let $\mathcal{FC}$ be one of* #P, GapP, *and* SpanP. *Then $\mathcal{FC}$ contains a parsimonious complete one-to-one function if and only if* $\mathrm{P} = \mathrm{PP}$.

Proof: Note that $\#\mathrm{P} \subseteq \mathrm{GapP}$ and $\#\mathrm{P} \subseteq \mathrm{SpanP}$. We will first show that if one of #P, GapP, and SpanP contains a parsimonious complete one-to-one function then $\mathrm{P} = \mathrm{C}_{=}\mathrm{P}$, and so, by Fact 2.1, $\mathrm{P} = \mathrm{PP}$.

Let $\mathcal{FC}$ be one of #P, GapP, and SpanP, and let g be $\mathcal{FC}$ parsimonious complete one-to-one function. Let L be an arbitrary $\mathrm{C}_{=}\mathrm{P}$ language. By definition, there is a polynomial p and a polynomial-time predicate R such that

$$x \in L \iff \|\{y \mid |y| = p(|x|) \wedge R(x, y)\}\| = 2^{p(|x|)-1}.$$

Let f be a function that given input x returns the number of words y of length $p(|x|)$ such that $R(x, y)$ holds. Clearly, f is a #P function, and thus it is parsimonious reducible to g via some function φ. Naturally, the function h defined for all x by $h(x) = 2^{p(|x|)-1}$ is parsimonious reducible to g. Let ψ be a function that acts as a parsimonious reduction from h to g.

For all x, we have $x \in L$ if and only if $f(x) = h(x)$. However, $f(x) = h(x)$ if and only if $g(\varphi(x)) = g(\psi(x))$, and since g is one-to-one, we have that $x \in L$ if and only if $\varphi(x) = \psi(x)$. This gives a test for L that can be done in polynomial time. Thus, $L \in \mathrm{P}$. Since L was chosen as an arbitrary language in $\mathrm{C}_{=}\mathrm{P}$, we have that $\mathrm{P} = \mathrm{C}_{=}\mathrm{P}$ and so, by Fact 2.1, $\mathrm{P} = \mathrm{PP}$.

Now we need to show that if $\mathrm{P} = \mathrm{PP}$ then for each $\mathcal{FC}$ in $\{\#\mathrm{P}, \mathrm{GapP}, \mathrm{SpanP}\}$ it holds that $\mathcal{FC}$ contains a parsimonious complete one-to-one function.

Let us first observe that if $\mathrm{P} = \mathrm{PP}$ then, by Fact 2.2, $\#\mathrm{P} = \mathrm{SpanP}$. We observe that if $\mathrm{P} = \mathrm{C}_{=}\mathrm{P}$ then $\#\mathrm{P} \subseteq \mathrm{FP}$. Let f be some #P function and p be a polynomial such that for every word x it

holds that $f(x) \leq 2^{p(|x|)}$. Since P = C$_=$P, it is easy to see that the following two languages, L' and L'', are both in P:

$$\begin{aligned} L' &= \{\langle x, y\rangle \mid f(x) = y\} \\ L'' &= \{\langle x, i, b\rangle \mid i^{th} \text{ bit of the integer } f(x) \text{ is } b\}. \end{aligned}$$

L' is in P because it clearly is in C$_=$P and we have P = C$_=$P. Also, L'' is in P because it is accepted by a nondeterministic polynomial-time Turing machine N that on input $\langle x, i, b\rangle$ guesses a binary encoding of an integer $y \leq 2^{p(x)}$, tests if $f(x) = y$ (by checking if $\langle x, y\rangle \in L'$), and if so, accepts if the i^{th} bit of y is b and rejects otherwise. This proves that $L'' \in$ NP and thus, by our assumption, $L'' \in$ P.

Now, to compute $f(x)$ in polynomial time, it is sufficient to query the language L'' about (at most) $p(|x|)$ bits of the value $f(x)$. Thus, we have that #P $\subseteq$ FP. Clearly, FP contains a parsimonious complete one-to-one function that belongs to #P, namely, the identity function.

By a similar argument, we can get a handle on the value of each GapP function. This is because each GapP function is a subtraction of two #P functions, and thus, a subtraction of two FP$_t$ function. As a result, the following function k is GapP parsimonious-complete and one-to-one.

$$k(x) = \begin{cases} \text{rank}(y) + 1 & \text{if } x = 0y, \\ -\text{rank}(y) - 1 & \text{if } x = 1y, \\ 0 & \text{if } x = \varepsilon. \end{cases}$$

This completes the proof. ■

Theorem 4.11 says that under reasonable assumptions there are no trivial reasons why #P parsimonious-complete functions shouldn't be parsimonious autoreducible.

As a corollary to the proof of Theorem 4.11 we have Corollary 4.12.

Corollary 4.12 *Both* TotP *and* #PE *contain a parsimonious complete one-to-one function if and only if* P = PP.

Proof: In the proof of Theorem 4.11 we have seen that if P = PP then, under the $\mathbb{N}$ semiring interpretation, FP$_t$ = #P. Since both TotP and #PE are subsets of #P, if P = PP then FP$_t$ = TotP = #PE and thus both of them have parsimonious-complete one-to-one functions.

As shown in the proof of Theorem 4.11, to show that if either TotP or #PE has a one-to-one parsimonious-complete function f then P = PP it is enough to show that having such a function f allows one to test equality of arbitrary #P functions in polynomial time.

Let us first consider the case of #PE. Let g and h be two arbitrary #P functions. We define $g'(x) = g(x) + 1$ and $h'(x) = h(x) + 1$. Clearly, both g' and h' are in #PE. Let ψ_g be an FP$_t$ function reducing g' to f and let ψ_h be an FP$_t$ function that reduces h' to f. Clearly we have that $g(x) = h(x)$ if and only if $g'(x) = h'(x)$. This is equivalent to $f(\psi_g(x)) = f(\psi_h(x))$. Since f is one-to-one, this is equivalent to $\psi_g(x) = \psi_h(x)$. This test can clearly be performed in polynomial time.

The case of TotP is handled very similarly. It is implicit in the paper of Kiayias et al. [KPSZ98] that for any two #P functions g and h there are two TotP functions g' and h' and a polynomial p such that $g(x) = g'(x) - 2^{p(|x|)}$ and $h(x) = h'(x) - 2^{p(|x|)}$. Having g' and h', we proceed as in the case of #PE. ■

5 Open Questions

There are several open questions that naturally come out from the work presented in this paper. In Section 3 we have shown that for many natural classes of functions and languages it is not the case that all their complete elements are length-decreasing self-reducible. It is natural to ask about other types of self-reducibility, not only the length-decreasing one. For example, can we show that all NP-complete languages are self-reducible via a word-decreasing self-reducibility? In general, are all NP-complete languages self-reducible in the sense of Meyer and Paterson? For the case of word-decreasing self-reductions, one might want to focus on a version that requires that if one runs the self-reduction algorithm recursively (instead of querying the oracle) then the depth of the recursion is only polynomial in the size of the input.

A very natural open question, though challenging, is whether parsimonious complete #P functions have parsimonious autoreductions. If we were able to show that this is the case, we would prove that #P is different from FP_{t}.

Regarding the discussion of algebraic properties of #P parsimonious-complete function in the end of Section 4, a long-standing open problem posed by Hemaspaandra and Ogihara [OH93] asks if #P is closed under proper decrement.

Finally, a very natural question to ask is whether all PP-complete languages are autoreducible. The reason to ask this question is that PP is closely related to #P and one could hope that some results regarding #P would translate to PP. Feigenbaum and Fortnow [FF91] showed that PP-complete languages are random-self-reducible. We are interested if one can establish deterministic autoreducibility, perhaps even in a many-one fashion.

Acknowledgments

We wish to thank Daniel Pierre Bovet and Pierluigi Crescenzi for inspiring this work. The results regarding length-decreasing self-reducibility presented in this paper were motivated by Problem 5.15 in their textbook [BC93]. The problem asks to prove that SAT is length-decreasing self-reducible and then asks whether this fact is enough for us to conclude that *all* NP-complete sets are length-decreasing self-reducible. This paper gives a solution to this problem. We thank Lane Hemaspaandra, Staszek Radziszowski and Alan Selman for useful discussions. We would also like to thank Aris Pagourtzis for interesting discussions, useful comments, and bringing the self-reducibility of functions to our attention. This work is supported in part by NSF Grants EIA-0080124, EIA-0205061, and CCF-0426761.

References

[AFK89] M. Abadi, J. Feigenbaum, and J. Kilian. On hiding information from an oracle. *Journal of Computer and System Sciences*, 39(1):21–50, 1989.

[Amb83] K. Ambos-Spies. P-mitotic sets. In *Logic and Machines*, pages 1–23, 1983.

[Bal90] J. Balcázar. Self-reducibility. *Journal of Computer and System Sciences*, 41(3):367–388, 1990.

[BC93] D. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity.* Prentice-Hall, 1993.

[BCP83] A. Borodin, S. Cook, and N. Pippenger. Parallel computation for well-endowed rings and space-bounded probabilistic machines. *Information and Control*, 58(1–3):113–136, 1983.

[BF92] R. Beigel and J. Feigenbaum. On being incoherent without being very hard. *Computational Complexity*, 2(1):1–17, 1992.

[BFvMT00] H. Buhrman, L. Fortnow, D. van Melkebeek, and L. Torenvliet. Separating complexity classes using autoreducibility. *SIAM Journal on Computing*, 29(4):1497–1520, 2000.

[BT96] H. Buhrman and L. Torenvliet. P-selective self-reducible sets: A new characterization of P. *Journal of Computer and System Sciences*, 53(2):210–217, 1996.

[FF91] J. Feigenbaum and L. Fortnow. On the random-self-reducibility of complete sets. In *Proceedings of the 6th Structure in Complexity Theory Conference*, pages 124–132. IEEE Computer Society Press, June/July 1991.

[FFK94] S. Fenner, L. Fortnow, and S. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48(1):116–148, 1994.

[FFLS92] J. Feigenbaum, L. Fortnow, C. Lund, and D. Spielman. The power of adaptiveness and additional queries in random-self-reductions. In *Proceedings of the 7th Structure in Complexity Theory Conference*, pages 338–346. IEEE Computer Society Press, June 1992. Final version appears in *Computational Complexity*, v. 4, pp. 158–174, 1994.

[FKN90] J. Feigenbaum, S. Kannan, and N. Nissan. Lower bounds on random-self-reducibility. In *Proceedings of the 5th Structure in Complexity Theory Conference*, pages 100–109. IEEE Computer Society Press, July 1990.

[Gil77] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.

[GOP+05] C. Glaßer, M. Ogihara, A. Pavan, A.L. Selman, and L. Zhang. Autoreducibility, mitoticity, and immunity. Technical Report TR05-011, Electronic Colloquium on Computational Complexity, http://www.eccc.uni-trier.de/eccc/, January 2005.

[Gup92] S. Gupta. On the closure of certain function classes under integer division by polynomially bounded functions. *Information Processing Letters*, 44(2):205–210, 1992.

[Gup95] S. Gupta. Closure properties and witness reduction. *Journal of Computer and System Sciences*, 50(3):412–432, 1995.

[HO02] L. Hemaspaandra and M. Ogihara. *The Complexity Theory Companion*. Springer-Verlag, 2002.

[JY85] D. Joseph and P. Young. Some remarks on witness functions for non-polynomial and non-complete sets in NP. *Theoretical Computer Science*, 39(2–3):225–237, 1985.

[KPSZ98] A. Kiayias, A. Pagourtzis, K. Sharma, and S. Zachos. The complexity of determining the order of solutions. In *Proceedings of the First Southern Symposium on Computing*, December 1998. http://pax.st.usm.edu/cmi/fscc98.html.

[Kre88] M. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36(3):490–509, 1988.

[KST89] J. Köbler, U. Schöning, and J. Torán. On counting and approximation. *Acta Informatica*, 26(4):363–379, 1989.

[KW98] J. Köbler and O. Watanabe. New collapse consequences of NP having small circuits. *SIAM Journal on Computing*, 28(1):311–324, 1998.

[Lad73] R. Ladner. Mitotic recursively enumerable sets. *Journal of Symbolic Logic*, 38(2):199–211, 1973.

[Lip91] R. Lipton. New directions in testing. In J. Feigenbaum and M. Merritt, editors, *Distributed Computing and Cryptography*, pages 191–202. DIMACS series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1991.

[LL76] R. Ladner and N. Lynch. Relativization of questions about log space computability. *Mathematical Systems Theory*, 10(1):19–32, 1976.

[LLS75] R. Ladner, N. Lynch, and A. Selman. A comparison of polynomial time reducibilities. *Theoretical Computer Science*, 1(2):103–124, 1975.

[MP79] A. Meyer and M. Paterson. With what frequency are apparently intractable problems difficult? Technical Report MIT/LCS/TM-126, Laboratory for Computer Science, MIT, Cambridge, MA, 1979.

[OH93] M. Ogiwara and L. Hemachandra. A complexity theory for feasible closure properties. *Journal of Computer and System Sciences*, 46(3):295–325, 1993.

[Pag01] A. Pagourtzis. On the complexity of hard counting problems with easy decision version. In *Proceedings of the 3rd Panhellenic Logic Symposium*, July 2001.

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

[PZ06] A. Pagourtzis and S. Zachos. The complexity of counting functions with easy decision version. In *Proceedings of the 31st International Symposium on Mathematical Foundations of Computer Science*, pages 741–752. Springer-Verlag *Lecture Notes in Computer Science #4162*, August/September 2006.

[Sch76] C. Schnorr. Optimal algorithms for self-reducible problems. In *Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming*, pages 322–337. Edinburgh University Press, July 1976.

[Sim75] J. Simon. *On Some Central Problems in Computational Complexity*. PhD thesis, Cornell University, Ithaca, N.Y., January 1975. Available as Cornell Department of Computer Science Technical Report TR75-224.

[Tor91] J. Torán. Complexity classes defined by counting quantifiers. *Journal of the ACM*, 38(3):753–774, 1991.

[Val79] L. Valiant. The complexity of computing the permanent. *Theoretical Computer Science*, 8(2):189–201, 1979.

[Vol94] H. Vollmer. On different reducibility notions for function classes. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, pages 449–460. Springer-Verlag *Lecture Notes in Computer Science #775*, February 1994.

[Wag86] K. Wagner. The complexity of combinatorial problems with succinct input representations. *Acta Informatica*, 23(3):325–356, 1986.

[WT92] O. Watanabe and S. Toda. Polynomial time 1-Turing reductions from #PH to #P. *Theoretical Computer Science*, 100(1):205–221, 1992.

[Yao90] A. Yao. Coherent functions and program checkers. In *Proceedings of the 22nd ACM Symposium on Theory of Computing*, pages 84–94. ACM Press, May 1990.

[Zan91] V. Zankó. #P-completeness via many-one reductions. *International Journal of Foundations of Computer Science*, 2(1):76–82, 1991.