Electronic Colloquium on Computational Complexity, Report No. 28 (2003)

PSPACE Contains Almost Complete Problems

Olivier Powell *

Abstract

An almost complete set A for a complexity class $\mathcal{C}$ is a language of $\mathcal{C}$ which is not complete, but that has the property that "many" languages of $\mathcal{C}$ reduce to A, where the term "many" is used in reference to Lutz's resource bounded measure ($\mathcal{RBM}$). The question of the existence of almost complete sets is unanswered for small complexity classes, which are those that do not or are not known to contain $\mathtt{E} = \text{time}(2^{\mathcal{O}(n)})$. One of the reasons for the emptiness of quantitative-completeness results for small complexity class, as opposed to the case of big complexity classes, where such results are abundant, is the fact that Lutz's $\mathcal{RBM}$ does not work well for small classes. We use a variation of Lutz's $\mathcal{RBM}$ designed to work for small complexity classes from [AS94], and use a diagonalisation process from [ASMRT00] to prove that there exists a problem which is almost complete for PSPACE, the class of space efficiently decidable problems.

1 Introduction

The study of quantitative-completeness notions started in [Lut95], through the definition of weak completeness. A language L is weakly complete for a class $\mathcal{C}$ if it is a member of $\mathcal{C}$ and if the set $P_r(L)$ of languages of $\mathcal{C}$ that reduce to it (where r is a certain type of reduction, such as Turing reduction, manyone reduction, etc...), is not small, i.e not of null Lutz's resource bounded measure ($\mathcal{RBM}$), which was introduced in [Lut92]. Since then, quantitative-completeness has been intensively studied, in different classes, and under different reductions. A state of the art of quantitative-completeness results can be found at the end of [ASMRT00]. These results have contributed to the general study of the quantitative structure of big complexity classes, which contain E, problems decidable in time $2^{\mathcal{O}(n)}$. A survey of these results can be found in [Lut97].

Whereas many quantitative-completeness results have been gathered in big complexity classes, there are no such results for the so called small complexity classes, which do not, or are not known to contain E, such as P, the class of (time) efficiently decidable problems. One of the reasons is that Lutz's $\mathcal{RBM}$ does not work for small complexity classes. In order to obtain quantitative results for small complexity classes, efforts have been deployed to construct measure concepts applying to them. In [May94b], a notion of measure for PSPACE is constructed using a concept of *plogon* machines. Perhaps more successfully, two different measure concepts for P arose from the series of papers [AS94], [AS95] and [Str97]. ([Str97] is revisited in [Pow02], analysing and correcting a mistake in it). These constructions have enough explanatory power to have yielded results such as the proof in [CSS97] that the Lutz hypothesis, which is a strengthening of the $\mathtt{P} \neq \mathtt{NP}$ hypothesis, does not hold in small complexity classes. (For more details about the Lutz hypothesis, see [May94a], [AS94], [LM94], [JL95a], [LM96], [Lut96] or, for a survey of the previous results, [Lut97]).

We shall be concerned with obtaining a quantitative-completeness result for the class PSPACE. To do so, we add structure to PSPACE by defining a measure for it, adapting the construction for P from [AS94]. We then construct an almost complete set, under logspace-manyone reductions, using a diagonalisation process defined and used in [ASMRT00] to obtain similar results for the class E.

*Université de Genève, Centre Universitaire d'Informatique, rue du Général Dufour 24, CH-1211 Genève 4, Switzerland, olivier.powell@cui.unige.ch

ISSN 1433-8092

2 Preliminaries

2.1 Computational Model

While defining a measure for PSPACE in section 2.3, we shall have to be concerned with Turing Machines ($\mathcal{TM}$) computing in bounded space complexity, which are restricted in the way they can query their input. We follow the usual convention of providing these machines with random access to their input; that is, these machines have a special address tape, and if an integer i is written in binary on this tape, the machine can in unit time move its head to position i of its input tape. These machines are also given an output tape, which is write only, and no restriction is imposed on the size of the output: only the working space is bounded. What kind of elementary arithmetics can we perform with such machines? This question is usually avoided, perhaps because it is considered trivial. We do not agree with this point of view, since in many publications we admit that when looking in detail at some arithmetical operation performed by machines bounded logarithmically in working space or time, we could not convince ourselves that these operations could be carried out efficiently, with the choice made for representing rational numbers, (such as diadic rational numbers). For this reason, we have taken care to look in detail at the way rational numbers should be represented, in order to allow arithmetical operations to be computed easily. Our conclusion is that the representation chosen in definition 2.1, which was suggested in [Mos02], permits to compute basic arithmetics efficiently, as we show in section 2.2.

Definition 2.1 *A transducer $T : \{0,1\}^* \to \{0,1\}^* \times \{0,1\}^*$ is said to compute a rational function $f : \{0,1\}^* \to \mathbb{Q}$ if $\forall x \in \{0,1\}^*$ the output (a,b) of T, interpreted as a pair of integers coded in binary satisfies $f(x) = \frac{a}{b}$.*

Suppose that we have m functions f_i, $1 \leq i \leq m$, each of them being efficiently space computable. How efficiently can we compute easy arithmetics, such as the sum $f = \sum_1^m f_i$? The next section is dedicated to answering this question, through lemma 2.7. Informally, lemma 2.7 says the following.

Lemma 2.1 *If a family $\{f_i\}_{1 \leq i \leq m}$ of rational functions can be computed space efficiently, then elementary arithmetics can be computed on this family with only a logarithmic in m loss of efficiency.*

2.2 Adding and Multiplying with Small Space Complexity

2.2.1 Adding Natural Numbers

Suppose we have a family of functions $\{f_i\}_{1 \leq i \leq m}$ where $f_i : \{0,1\}^* \to \mathbb{N}$, such that each f_i is computable in small space complexity. How efficiently is it possible to compute the function $f := \sum_{i=1}^m f_i$? The next lemma states that it is possible to compute f with only a logarithmic in m loss of efficiency.

Definition 2.2 *Let y be an integer, we note $\mathtt{BIN}(y)$ the binary representation of y. Let f be a function, $f : \{0,1\}^* \to \mathbb{N}$. A Turing machine M is said to compute f if $\mathtt{BIN}(f(x)) = M(x)$.*

Lemma 2.2 *There exists a constant c such that for any $m \in \mathbb{N}$ and for any family of functions $\{f_i\}_{1 \leq i \leq m}$ with $f_i : \{0,1\}^* \to \mathbb{N}$, and such that $\forall i \; \exists M_i$ a Turing machine computing f_i in $\mathsf{SPACE} = g(n)$, (for a certain complexity function g), the function $f := \sum_{i=1}^m f_i$ is computable in $\mathsf{SPACE} = c[g(n) + \log m]$.*

Proof. For x with $|x| = n$, let $y_{il} \cdots y_{i1} := M_i(x)$, with $l = 2^{g(n)}$ (wlog). Let $A = A_{ij}$ with $A_{ij} = y_{ij}$, $1 \leq i \leq m$ and $1 \leq j \leq l$ be the table, which is dynamically computable in $\mathsf{SPACE} = g(n)$, and which is used to compute f according to the following standard way of adding numbers "by

hand":

$$\begin{array}{cccc} & y_{1l} & \cdots & y_{11} \\ & \vdots & & \vdots \\ + & y_{ml} & \cdots & y_{m1} \\ \hline & & \cdots & \end{array} \simeq A$$

Let $z_1 := \sum_{i=1}^{m} y_{i1}$, $z_2 := \sum_{i=1}^{m} y_{i2} + \lfloor \frac{z_1}{2} \rfloor$, ..., $z_m := \sum_{i=1}^{m} y_{im} + \lfloor \frac{z_{m-1}}{2} \rfloor$. Since for $1 \leq k \leq m$ $z_k \leq 2m$ and since table A can be dynamically computed in $\mathsf{SPACE} = g(n)$, each z_k can be computed and stored in memory using $\mathsf{SPACE} = g(n) + c[\log m + 1] = g(n) + c \log m$, for a certain constant c, (the last equality holds if $m > 1$, and for c replaced by another constant c', but assume this to be implicit). (Notice that while computing dynamically the table A, it will be necessary to compute and store indexes i and j ranging from 1 to l. Since $l \leq 2^{g(n)}$ (through trivial space-time tradeoffs), such indexes are stored in $\mathsf{SPACE} = g(n)$, and are thus absorbed in the constant c). Now to finish the proof, it only needs to be observed that computing z_1 and outputting its rightmost bit (corresponding to units), then computing z_2 and outputting its rightmost bit, ..., computing z_{m-1} and outputting its rightmost bit, and computing and outputting *every* bits of z_m produces the desired output. □

2.2.2 Multiplying Natural Numbers

Let f_1 and f_2 be two functions from $\{0,1\}^*$ to $\mathbb{N}$, such as in lemma 2.2 of the previous subsection. The following table permits to compute $f_1 \cdot f_2$, the product of the two functions:

$$\begin{array}{cccccc} & & y_{1l} & \cdots & y_{11} & \\ & \times & y_{2l} & \cdots & y_{21} & \\ \hline A_{1(2l)} & \cdots & \cdots & \cdots & A_{11} & \\ \vdots & & & & \vdots & \\ + \quad A_{l(2l)} & \cdots & \cdots & \cdots & A_{l1} & \\ \hline & & \cdots & & & \end{array} \simeq A, \quad \begin{array}{l} \text{where } A_{ij} = y_{2i} y_{1(j-i+1)}, \\ \text{with } y_{ik} = 0 \text{ if } k < 0 \text{ or } k > l. \end{array}$$

Let $h_i(x) := A_{i(2l)\cdots A_{i1}} \in \{0,1\}^{2l}$, for $1 \leq i \leq l$. Since y_{ij}'s can be computed in $\mathsf{SPACE} = g(n)$, and since the indexes range up to $2l$ (and can thus be stored on $g(n) + 1$ bits), the functions h_i can be computed in $\mathsf{SPACE} = c[g(n) + 1]$, for a certain constant c not depending on f_1 and f_2. Now to compute the value of $f = f_1 + f_2$, we could add the values of table A column-wise. To bound the space complexity of the computation of this sum, we can use lemma2.2, (replacing the f_i's of the lemma with the h_i's from above). The following corollary is then obtained:

Corollary 2.3 *$\exists c$ a constant such that if f_1 and f_2 are functions from $\{0,1\}^*$ to $\mathbb{N}$ computable in $\mathsf{SPACE} = g(n)$, then $f_1 \cdot f_2$ is computable in $\mathsf{SPACE} = c[g(n) + 1 + \log(l)] = c[g(n)]$.*

Using this corollary, an upper bound of $c^2 g(n)$ on the space required for computing $(f_1 \cdot f_2) \cdot f_3$ is obtained (we assumed for ease of notations that the constant term "+1" can be absorbed in the constant c). An easy induction argument gives the following corollary.

Corollary 2.4 *There exists a constant c such that for any $m \in \mathbb{N}$ and for any family of functions $\{f_i\}_{1 \leq i \leq m}$ with $f_i : \{0,1\}^* \to \mathbb{N}$, and such that $\forall i \; \exists M_i$ a Turing machine computing f_i in $\mathsf{SPACE} = g(n)$, (for a certain complexity function g), the function $f := \Pi_{i=1}^{m} f_i$ is computable in $\mathsf{SPACE} = c^{m-1} g(n)$.*

This exponential bound on the growth of the size required to compute the product of functions computing natural numbers is quite large. We shall next explain how it is possible to compute products of functions in a much more space efficient way than the one described above. The bound obtained will be logarithmic in the number m of functions to be multiplied. This is much better (twice logarithmically better) then the previous bound, which is exponential in m, and it shows that a bound nearly as good (up to multiplication by a constant factor) as the one from lemma 2.2 for additions holds for multiplications.

2.2.3 Improving the Space Efficiency of Multiplications

Lemma 2.5 *There exists a constant c such that for any functions f_1 and f_2 from $\{0,1\}^*$ to $\mathbb{N}$, computable by Turing machines M_1 (and M_2 respectively), in* $\mathsf{SPACE} = g_1(n)$ *(and* $\mathsf{SPACE} = g_2(n)$ *respectively), and such that $|M_1(x)| \leq l_1$ and $|M_2(x)| \leq l_2$ (i.e. $f_i(x) < 2^{l_i}$, where l_i is really $l_i(|x|)$), then $f_1 \cdot f_2$ is computable in* $\mathsf{SPACE} = \max\{g_1(n), g_2(n)\} + c\max\{\log l_1, \log l_2\}$.

Proof. Suppose wlog that $l_1 \geq l_2$, and consider the following multiplication table:

$$\begin{array}{rccccc} & y_{1l_1} & \cdots & \cdots & y_{1l_2} \\ \times & & y_{2l_2} & \cdots & y_{2l_2} \\ \hline A_{1(l_1+l_2)} & \cdots & \cdots & \cdots & A_{11} \\ \vdots & & & & \vdots \\ + \quad A_{l_2(l_1+l_2)} & \cdots & \cdots & \cdots & A_{l_2 1} \\ \hline & & \cdots & & \end{array} \simeq A, \quad \begin{array}{l} \text{where } A_{ij} = y_{2i}y_{1(j-i+1)}, \\ \text{with } y_{1k} = 0 \text{ if } k < 0 \text{ or } k > l_1. \end{array}$$

This table can be dynamically computed in $\mathsf{SPACE} = \max\{g_1(n), g_2(n)\}$, plus a space of size $\mathsf{SPACE} = c\max\{\log l_1, \log l_2\}$ (for a certain constant c) to store the indexes of the table. Once again the z_i's being the row-wise sum of table A have to be computed, and this takes $\mathsf{SPACE} = \log(2l_2)$ (c.f proof of lemma 2.2 for the details, which remain the same). □

Using this lemma, we can improve on corollary 2.4:

Lemma 2.6 *There exists a constant c such that for any $m \in \mathbb{N}$ and for any family of functions $\{f_i\}_{1\leq i\leq m}$ with $f_i : \{0,1\}^* \to \mathbb{N}$, and such that $\forall i\ \exists M_i$ a Turing machine computing f_i in* $\mathsf{SPACE} = g(n)$*, (for a certain complexity function g), the function $f := \Pi_{i=1}^m f_i$ is computable in* $\mathsf{SPACE} = c[g(n) + \log m]$.

Proof. The proof is obtained by induction. First let us define $\tilde{f}_0 := 1$ and $\tilde{f}_i := f_i\tilde{f}_{i-1}$ (for i from 1 to m). Notice that if we note $\tilde{l}_i$ to be the length of the output of a Turing machine computing f_i, then $\tilde{l}_i \leq il$, where $l \leq 2^{g(n)}$. Now let $E(i)$ be the space required to compute $\tilde{f}_i$. $E(1)$ has value $g(n)$ by hypothesis. Using lemma 2.5, we can bound $E(i)$ by $\max\{E(i-1), g(n)\} + c\max\{\log l, \log \tilde{l}_{i-1}\}$, which is equal to $E(i-1) + c\log \tilde{l}_{i-1} \leq E(i-1) + c\log(l(i-1))$. Solving the recursion gives $E(m) = g(n) + c[\log l + \log 2l + \cdots + \log(l(m-1))] = g(n) + c[\log(l(m-1))^{\frac{m}{2}}] = c[g(n) + \log m]$. □

2.2.4 Rational Functions and Arithmetics

It follows from the results of the two previous subsections that:

Lemma 2.7 *$\exists c$ a constant such that for any family of rational functions $\{f_i\}_{1\leq i\leq m}$ such that $\forall i$ f_i is computable in* $\mathsf{SPACE} = g(n)$*, (where n is the size of the input), simple arithmetics (sums, substractions, multiplications and divisions) can be carried out in* $\mathsf{SPACE} = c[g(n) + \log m]$

Proof. This holds because adding rational function requires multiplying the denumerators, and adding the (weighted) numerators, multiplication requires multiplying numerators and denumerators, which can be computed efficiently, by lemmas 2.2 and 2.6. □

2.3 A Measure on PSPACE

A measure for P which shares many of the properties of Lutz's $\mathcal{RBM}$, [Lut92], is defined in [AS94]. As pointed out in their paper, this measure for P can be adapted to PSPACE. We find it necessary to give a full description of this construction, including proofs. The reason is that the original definition of [AS94] is nearly a decade old, and $\mathcal{RBM}$ has evolved considerably since then, permitting clearer and cleaner definitions. The construction of [AS94] shares with Lutz's $\mathcal{RBM}$ the use of martingales as a central tool. Martingales are functions which can be interpreted intuitively as functions describing the course of the so-called "casino game", opposing a casino

and a gambler. While this approach is enlightening from the intuitive point of view, we do not describe it in this article, and refer the reader to the existing literature on the subject, e.g [Lut97] or [ASMRT00] for more details. Formally, martingales are thus defined.

Definition 2.3 *A super-martingale is a function $d : \{0,1\}^* \to \mathbb{R}^+$ such that $d(\omega) \geq \frac{d(\omega 0)+d(\omega 1)}{2}$, and a martingale is a super-martingale which satisfies the inequality above with the "$\geq$" replaced by an equality.*

As explained above, martingales are used to play a game, called the casino game. When the casino chooses a language L and the martingale "wins" against the casino, the language is said to be in the success set, or to be covered by the martingale. Before we give the formal definition of a success set, let us make the following convention.

Convention 1 *We put on $\{0,1\}^*$ the canonical lex-length ordering, and the symbol s_N always denotes the Nth word under this ordering.*

Definition 2.4 *The success set of a (super)-martingale d, is $S^\infty[d] := \{L \in \mathcal{P}(\{0,1\}^*) \mid \limsup_{N\to\infty} d(\chi_L[0,N]) = \infty\}$, where $\chi_L[0,N]$ is the prefix of length $N+1$ of the characteristic sequence of L under the canonical ordering.*

In order to develop an $\mathcal{RBM}$ for a class $\mathcal{C}$, we need to bound the resources available to compute martingales. How many restrictions we need to put on the resources available depends on the class for which we want to define a measure, which is PSPACE in the case we are interested in. There are two resources that we will need to consider. The first one is the common space complexity. The second resource is less usual, and is related to the way algorithms (or transducers) computing a martingale query their input, (remember we are using a model of transducers having RAM access to their input, c.f. section 2). To bound the query complexity of transducers in a way that suits our needs, we need the following definition of dependency sets.

Definition 2.5 *A dependency set is a family $\{G_N\}_{N\in\mathbb{N}}$ of subsets of $\{0,1\}^*$, such that the following holds: 1) $\forall N \in \mathbb{N}$, $G_N \subseteq \{s_0, \ldots s_N\}$. 2) $\exists k$ such that $\forall N \in \mathbb{N}$, $|G_N| \leq \log^k(N)+k$. 3) $\forall M \leq N$, it holds that $G_M \subseteq G_N$. A transducer T is said to query its input in a dependency set if the family $Q_N := \{s_i \mid \exists x \in \{0,1\}^N$ such that $T(x)$ queries the ith bit of its input during its computation $\}$ is such that $\{Q_N\}_{N\in\mathbb{N}}$ is a dependency set.*

With this definition at hand, we define "good" transducers (or Γ transducers) by limiting the space complexity of computations, and requiring the queries to the input to be done inside a dependency set.

Definition 2.6 *A transducer T is a Γ(PSPACE) transducer if on an input of length N, T computes in $\mathsf{SPACE} = \mathcal{O}(\log N)$ querying its input in a dependency set.*

The next convention is a detail, but we will need it in our effort to rigorously construct an $\mathcal{RBM}$ for PSPACE. Its meaning will become clear when it is used. Intuitively, we sometimes need to consider transducers which have an auxiliary input tape, on which the length of the (main) input is written in binary. Let T be such a transducer, the convention that for any input x of length N, the transducer $T(x)$ has the value N given as input on an auxiliary tape is explicitly shown using the following notational convention.

Convention 2 *Let T be a transducer. We write $T_N(x)$ (or T_N) instead of $T(x)$ (or T respectively) when T_N is a Γ(PSPACE) transducer supplied with the length of its input on an auxiliary tape.*

Note that a transducer can very easily compute the length of its input. (It can do so in logarithmic time, c.f. [Bus87]). The problem that we want to avoid by supplying a transducer T with N on an auxiliary tape, is the way T queries its input. Therefore, no restriction is implied on the way $T_N(x)$ queries its auxiliary tape: only x is queried "according" to a dependency set. The definition of "good" functions and martingales follows.

Definition 2.7 *A* $\Gamma(\mathtt{PSPACE})$ *function is a function which is computed by a* $\Gamma(\mathtt{PSPACE})$ *transducer* T. *A* $\Gamma(\mathtt{PSPACE})$ *martingale is a martingale which is computed by a* $\Gamma(\mathtt{PSPACE})$ *transducer* T_N.

We now have all that is required to define a measure μ on PSPACE. μ is a partial function from subsets of PSPACE to $\{0,1\}$, which is thus defined.

Definition 2.8 (A measure for PSPACE) *The function* $\mu : \mathcal{P}(\mathtt{PSPACE}) \dashrightarrow \{0,1\}$ *is defined by* $\mu(\mathcal{A}) = 0$ *if* $\exists d$ *a* $\Gamma(\mathtt{PSPACE})$ *martingale such that* $\mathcal{A} \subseteq S^{\infty}[d]$, *and* $\mu(\mathcal{A}) = 1$ *if* $\mu(\mathtt{PSPACE} \setminus \mathcal{A}) = 0$.

Intuitively, this measure defines a notion of big sets, (those of measure 1), and a notion of small sets, (those of measure 0). The fact that this definition is consistent, (i.e. there are no subsets $\mathcal{A}$ of PSPACE such that $\mu(\mathcal{A}) = 0$ and $\mu(\mathcal{A}) = 1$ at the same time), is an easy consequence (left to the reader) of the measure conservation lemma (lemma 2.8). The terminology of calling μ a measure is justified by showing that μ satisfies the following "measure axioms" which we shall make precise.

M1 Easy unions of null sets are null sets (lemma 2.12).

M2 Singleton sets are null sets (lemma 2.9).

M3 PSPACE itself is not a null set (lemma 2.8).

We now state the lemmas asserting that the "measure axioms" hold, starting with the measure conservation lemma which insures that the biggest subset of PSPACE, which is PSPACE itself, "looks" big to μ, i.e. has measure 1.

Lemma 2.8 (M3, Measure Conservation) $\mu(\mathtt{PSPACE}) \neq 0$

Proof.(2.8) It has to be shown that no single $\Gamma(\mathtt{PSPACE})$ martingale covers the whole of PSPACE. To prove this, we show that for any $\Gamma(\mathtt{PSPACE})$ martingale d, there exists a language L of PSPACE which is not covered by d. Let d be a $\Gamma(\mathtt{PSPACE})$ martingale, and define recursively the language L by $L(s_0) = 0$, and for $s_N \in \{0,1\}^*$ the Nth word, ($N > 0$), $L(x) := 1$ if $d(\chi_L[0, N-1]0) \geq d(\chi_L[0, N-1]1)$, $L(x) = 0$ otherwise. First, notice that $L \not\subseteq S^{\infty}[d]$, since L "defeats" d at each single step. To conclude, it only remains to show that L is in PSPACE. Let us describe an algorithm A_L running in space polynomial in its input size, such that $A_L(x) = L(x)\ \forall x$. Suppose we have to compute $L(s_N)$, $n := |s_N| = \log N$. Trivially, our work would be finished if we could compute $d(\chi_L[0, N-1]0)$ and $d(\chi_L[0, N-1]1)$ in polynomial space in n. Let us show that this is the case. Since d is a $\Gamma(\mathtt{PSPACE})$ martingale, there exists a transducer T_N which, given $\chi_L[0, N-1]0$ as an input, computes $d(\chi_L[0, N-1]0)$ in polylogarithmic space in N, which is polynomial in n. We shall thus describe an algorithm M, running in polynomial space in its input size, which on input s_N computes $d(\chi_L[0, N-1]0)$. (Computing $d(\chi_L[0, N-1]1)$ is done similarly). First, notice that computing N from s_N is no problem to an algorithm computing in polynomial space, so the fact that T_N has to have access to N on an auxiliary tape is easily solved. The problem, while $M(s_N)$ tries to simulate the computation of $T_N(\chi_L[0, N-1]0)$ in polynomial space in n, is to enable the access to $\chi_L[0, N-1]$. Remember that we are describing an algorithm $A_L(s_N)$ which uses $M(s_N)$ to compute $L(s_N)$. We see that the algorithm M has to simulate T_N, which needs to have access to $\chi_L[0, N-1]0$. This raises the spectre of having to call A_L recursively on every inputs s_i, $0 \leq i < N$, (since $\chi_L[i] = A_L(s_i)$). This would be a recursion of depth N, thus exponential in the size of the input s_N, and which can thus not be afforded by $A_L(s_N)$, which we would like to let run in polynomial space l in n. Thanks to the fact that T_N is a $\Gamma(\mathtt{PSPACE})$ transducer, this is not the case: $T_n(\chi_L[0, N-1]0)$ only queries its input in a dependency set of size polylogarithmic in N, which is thus polynomial in n. This enables us to bound polynomially in the size of its input the recursive call of A_L to itself. □

Next, we state the lemma asserting that the second measure axiom holds. This axiom requires that the smallest (non empty) subsets of PSPACE "look" small to μ, that is singleton sets consisting of only one language of PSPACE should be of null measure.

Lemma 2.9 (M2) $\forall L \in \mathtt{PSPACE}$, $\mu(\{L\}) = 0$.

Proof.(2.9) Consider a martingale d which bets only on words of the form 0^n, for a certain $n \in \mathbb{N}$, and each time it wagers, it tries to double its capital, betting on the fact that it is playing against the language L. Since L is in PSPACE, d can be computed in polylogarithmic space in its input size , and since d only wagers on words of $\{0\}^*$, this computation can be done by only querying the input in a dependency set. (More precisely, the dependency set is $\{G_N\}_{N\in\mathbb{N}}$, where $G_k = \{0^l \mid 0 \leq l \leq k\}$). □

In order to prove lemma 2.12 stating that easy unions of null sets gives a null set again, we first need a few technical lemmas. The first lemma states that a $\Gamma(\mathtt{PSPACE})$ super-martingale can always be replaced by a $\Gamma(\mathtt{PSPACE})$ martingale.

Lemma 2.10 *Let d be a $\Gamma(\mathtt{PSPACE})$ super-martingale, then there exists $\hat{d}$ a $\Gamma(\mathtt{PSPACE})$ martingale such that $S^\infty[d] \subseteq S^\infty[\hat{d}]$.*

Proof.(2.10) To prove this lemma, it suffices to construct $\hat{d}$ a $\Gamma(\mathtt{PSPACE})$ martingale such that $\hat{d} \geq d$. First of all, let us define the following: for $\omega \in \{0,1\}^*$ and $b \in \{0,1\}$, we define $WIN(\omega b) := d(\omega b) - d(\omega)$, and for $b \in \{0,1\}$, we let $\bar{b} := 1 - b$. Notice that since d satisfies $d(\omega) \geq \frac{1}{2}[d(\omega b) + d(\omega\bar{b})]$, it holds that $WIN(\omega b) + WIN(\omega\bar{b}) = d(\omega b) + d(\omega\bar{b}) - 2d(\omega) \leq 0$, and thus $-WIN(\omega\bar{b}) \geq WIN(\omega b)$.

Let us now define $\hat{d}$ recursively: $\hat{d}(\lambda) := d(\lambda)$ (where $\lambda = s_0$ is the empty word), and $\hat{d}(\omega b) := \hat{d}(\omega) + \frac{1}{2}[d(\omega b) - d(\omega\bar{b})]$. In order to substantiate the proof, it remains to be shown that $\hat{d}$ is indeed a $\Gamma(\mathtt{PSPACE})$ martingale such that $\hat{d} \geq d$.

Let us start by showing that $\hat{d} \geq d$ by induction. First, notice: $\hat{d}(\omega b) \stackrel{!}{=} \hat{d}(\omega) + \frac{1}{2}[d(\omega b) - d(\omega\bar{b})] \stackrel{\text{induction}}{\geq} d(\omega) + \frac{1}{2}[d(\omega b) - d(\omega) + d(\omega) - d(\omega\bar{b})] = d(\omega) + \frac{1}{2}[WIN(\omega b) - WIN(\omega\bar{b})]$. Next, notice: $d(\omega b) = d(\omega) + WIN(\omega b)$. Thus, to show that $\hat{d} \geq d$, it suffices thus to show that: $WIN(\omega b) \leq \frac{1}{2}[WIN(\omega b) - WIN(\omega\bar{b})]$. But this is easily seen to hold from the fact (shown above), that $-WIN(\omega\bar{b}) \geq WIN(\omega b)$.

Next, notice that $\hat{d}$ is a martingale, since it immediately follows from the definition of $\hat{d}$ that $\hat{d}(\omega b) + \hat{d}(\omega\bar{b}) = 2\hat{d}(\omega)$.

Finally, to prove that $\hat{d}$ is a $\Gamma(\mathtt{PSPACE})$ martingale, remember that $\hat{d}(\omega b) := \hat{d}(\omega) + \frac{1}{2}[d(\omega b) - d(\omega\bar{b})]$. The right hand side of the sum is no problem, since d is by hypothesis a $\Gamma(\mathtt{PSPACE})$ super-martingale. It may be feared that the left hand side of the sum starts a recursion of depth $|\omega|$, but this is not the case since from the fact that d is a $\Gamma(\mathtt{PSPACE})$ martingale, it is possible to compute d querying the input only in a dependency set G_N, thus enabling to bound the depth of the recursion. More precisely, the following holds: $\hat{d}(\omega) = \frac{1}{2}\sum_{\{k \mid s_K \in G_{d,N}\}} d(\omega[0,k-1]\omega[k]) - d(\omega[0,k-1](1-\omega[k]))$, which permits to bound the depth of the recursion by $|G_N| = \log^k(n)$ (for a certain integer k). □

Next, we need another lemma, perhaps slightly more complicated, stating that if we have a martingale which lets itself be approximated by a $\Gamma(\mathtt{PSPACE})$ function, we can always replace this martingale by a $\Gamma(\mathtt{PSPACE})$ martingale covering the same subsets of PSPACE.

Lemma 2.11 (Exact computation) *Let d be a martingale. Let $\tilde{d}$ be a function such that there exists T_N a $\Gamma(\mathtt{PSPACE})$ transducer computing $\tilde{d}$, and such that for every ω, it holds that $|d(\omega) - \tilde{d}(\omega)| \leq \frac{1}{|\omega|}$. Then there exists $\hat{d}$, a $\Gamma(\mathtt{PSPACE})$ martingale such that $S^\infty[d] \subseteq S^\infty[\hat{d}]$.*

Proof.(2.11) We start by defining $\bar{d}$ a $\Gamma(\mathtt{PSPACE})$ super-martingale such that $S^\infty[d] \subseteq S^\infty[\bar{d}]$: for $\omega \in \{0,1\}^N$, $\bar{d}$ is defined by $\bar{d}(\omega) := \tilde{d}(\omega) + \frac{3}{2^n}$, where $n := \log N$.

First, we show that $\bar{d}$ is a super-martingale: $\bar{d}(\omega b) \stackrel{!}{=} \tilde{d}(\omega b) + \frac{3}{2^{n+1}} \leq d(\omega b) + \frac{3}{2^{n+1}} + \frac{1}{2^{n+1}} = 2d(\omega) - d(\omega\bar{b}) + \frac{4}{2^{n+1}} \Rightarrow \frac{\bar{d}(\omega b)+d(\omega\bar{b})}{2} \leq d(\omega) + \frac{2}{2^{n+1}} \Rightarrow \frac{\bar{d}(\omega b)+\tilde{d}(\omega\bar{b})}{2} \leq d(\omega) + \frac{2}{2^{n+1}} + \frac{1}{2^{n+2}} = d(\omega) + \frac{5}{2^{n+2}} \Rightarrow \frac{\bar{d}(\omega b)+\bar{d}(\omega\bar{b})}{2} - \frac{3}{2^{n+2}} \leq d(\omega) + \frac{5}{2^{n+2}} \leq \tilde{d}(\omega) + \frac{5}{2^{n+2}} + \frac{1}{2^n} = \tilde{d}(\omega) + \frac{9}{2^{n+2}} = \bar{d}(\omega) + \frac{9}{2^{n+2}} - \frac{12}{2^{n+2}} =$

$\bar{d}(\omega) - \frac{3}{2^{n+2}} \Rightarrow \frac{\bar{d}(\omega b)+\bar{d}(\omega \bar{b})}{2} \leq \bar{d}(\omega)$. To see that $\bar{d}$ is a $\Gamma(\texttt{PSPACE})$ super-martingale, we need to show that there exists $\bar{T}_N$ a $\Gamma(\texttt{PSPACE})$ transducer computing $\bar{d}$ by only querying its input in a dependency set $\{\bar{G}_N\}_{N\in\mathbb{N}}$. This latter fact is easily seen to follow from the fact that by hypothesis $\bar{d}$ is a $\Gamma(\texttt{PSPACE})$ martingale, and that lemma 2.7 guarantees that it is possible to add the $\frac{3}{2^n}$ term to $d(\omega)$. Finally, it trivially holds that $S^\infty[d] \subseteq S^\infty[\bar{d}]$, since $d \leq \bar{d}$. To finish the proof, we use lemma 2.10 to insure the existence of a $\Gamma(\texttt{PSPACE})$ martingale $\hat{d}$ such that $S^\infty[\bar{d}] \subseteq S^\infty[\hat{d}]$. □

In contrast with the two other measure axioms, the first one needs to be made precise, by explaining what is meant by an "easy" union of null sets. Roughly speaking, a union of null sets is easy if it is uniform, from the point of view of martingales, and if the union of its dependency sets is not too large. Precisely, the first measure axiom is the following condition under which the union of a family of null sets gives a null set again.

Lemma 2.12 (M1) *Let $\{X_i\}_{i\in\mathbb{N}}$ be a family of null sets, and let d_i be a family of martingales such that $X_i \subseteq S^\infty[d_i]$. Suppose there exists T_N a transducer such that: 1) $T_N(i,x) = d_i(x)$. 2) $\exists k \in \mathbb{N}$ such that $\forall i \in \mathbb{N}$, $T_N(i,\cdot)$ computes in $\mathsf{SPACE} = (i^k+k)(\log^k(N)+k)$, querying (the right hand side of) its input in a dependency set $\{G_{i,N}\}_{N\in\mathbb{N}}$ such that $|G_{i,N}| \leq (i^k+k)(\log^k(N)+k)$, then $\mu(\cup_{i\in\mathbb{N}}X_i) = 0$.*

Proof.(2.12) We are going to exhibit a $\Gamma(\texttt{PSPACE})$ martingale that covers $\cup_i X_i$. First of all, consider the martingale $d := \sum_{i=1}^\infty c_i d_i$, for a certain family of c_i's which can be computed (from i) in $\mathsf{SPACE} = i^k + k$, (for a certain constant k), and such that $\sum_i c_i = c$ for a certain constant c. It easy to see that d is a martingale such that $\cup X_i \subseteq S^\infty[d]$. It remains to be shown that d can be approximated by a $\Gamma(\texttt{PSPACE})$ function $\tilde{d}$. The exact computation lemma (lemma 2.11) will then permit to conclude. Let $\omega \in \{0,1\}^N$. $\tilde{d}$ is defined by: $\tilde{d}(\omega) := \sum_{i=1}^{n:=\log N} c_i d_i$. We now turn to the analysis of the complexity of computing $\tilde{d}$: the transducer T_N in the hypothesis of the lemma insures that d_i's are computable in polynomial space in i and polylogarithmic space in N, which is globally polylogarithmic in N, since $i \leq n = \log N$. The c_i's are by hypothesis computable in polynomial space in i, which is thus polylogarithmic in N too. Finally, lemma 2.7, insures that the arithmetics required to multiply and sum up the previous c_i's and d_i's can still be computed in polylogarithmic space in N. Thus, $\exists k \in \mathbb{N}$ and M_N a transducer computing $\tilde{d}$ in $\mathsf{SPACE} = \log^k(N)+k$. In order to prove that $\tilde{d}$ is a $\Gamma(\texttt{PSPACE})$ function, we still need to prove that M_N queries its input only in a set G_N such that $\{G_N\}_{N\in\mathbb{N}}$ is a dependency set. To prove this, notice that on input $\omega \in \{0,1\}^N$, the computation of $\tilde{d}$ by M_N queries its input in the following set: $\cup_{i\leq n} G_{i,N}$, the size of which is at most $n(i^k+k)(\log^k(N)+k) \leq (\log^k(N)+k)^3 = \log^{k'}(N)+k'$, and that since for any i, $\{G_{i,N}\}$ is a dependency set, it follows that $\{G_N\}$ is a dependency set too (i.e. $G_M \subseteq G_N$ if $M \leq N$). Thus $\tilde{d}$ can be computed by M_N a $\Gamma(\texttt{PSPACE})$ transducer. We next prove that, for a suitable choice of c_i's, $\tilde{d}$ is an approximation of d. Let define $c_i := \frac{1}{2^{2^i}2^i 2^i}$. First notice that c_i can be computed (from i) in polynomial space in i, as required in the construction above. Indeed, the only difficulty would be to compute 2^{2^i}, the rest is "simple" arithmetics (as proved in lemma 2.7). Notice that 2^{2^i} is coded (in binary) as 10^{2^i}. So to compute 2^{2^i}, the only thing an algorithm has to do is to output a 1, and then use a counter ranging from 1 to 2^i to output 0's. The space required for the computation of this algorithm is mainly the space needed to store the counter, which is i, and thus polynomial in i, as announced. Next, let us show that with this choice of c_i's, $\tilde{d}$ becomes an approximation of d: Since $d_i(\omega) \leq 2^N$ (assuming wlog that $d_i(\lambda) = 1\ \forall i$), $|\tilde{d}(\omega) - d(\omega)| = |\sum_{i=1}^n c_i d_i(\omega) - \sum_{i=1}^\infty c_i d_i(\omega)|$ is bounded by $\sum_{i=n}^\infty c_i 2^N \leq \sum_{i=n}^\infty \frac{2^N}{2^{2^i}}\frac{1}{2^i 2^i} \leq \frac{2^N}{2^N 2^n}\sum_{i=n}^\infty \frac{1}{2^i} \leq \frac{1}{2^n} = \frac{1}{N}$. The exact computation lemma, (lemma 2.11), permits to conclude that $\cup X_i$ is a null measure set. □

3 An Almost Complete Set for PSPACE

We want to use the definition of μ from the previous section to construct an almost complete set for PSPACE. This is a technical task which is quite complicated, and justifies splitting things into

subsections. We keep separate (in subsection 3.1) the general definitions and conventions from the main construction (in subsection 3.2). It is only in section 3.3 that we actually prove that this construction yields the existence of an almost complete set for PSPACE.

3.1 Definitions and Conventions

We shall set the definitions and conventions required in the rest of the article, *trying* to avoid unnecessarily heavy notations. For example, although different type of reductions exist (manyone, Turing, etc...), we define *reductions* as logspace-manyone reductions, since they are the only type of reductions we are concerned with in this paper. For completeness, we remind the reader of the definition of (logspace-manyone) reductions.

Definition 3.1 *A function* $R : \{0,1\}^* \to \{0,1\}^*$ *is a reduction from the language* A *to the language* B *(noted* $A \leq B$*) if* $\forall x \in \{0,1\}^*$, $x \in B$ *iff* $R(x) \in A$ *and* $\exists\ k \in \mathbb{N}$ *and* T *a transducer computing* R *in* $\mathsf{SPACE} = \mathcal{O}(\log n)$.

The (lower) span of language is the set of languages that reduce to it. Roughly speaking, a language is complete if its span contains the whole complexity class (which is PSPACE in the case we are interested in), and it is almost complete if its span is "large", from a measuring point of view.

Definition 3.2 *The span of a language* A *is* $P_m(A) := \{B \mid B \leq A\}$. *If* $A \in$ PSPACE, *it is* complete *if* $P_m(A) \supseteq$ PSPACE *and it is* almost complete *if (it is not complete) and that* $\mu(P_m(A) \cap$ PSPACE$) = 1$.

We have announced to the reader that the construction of an almost complete set (in section 3.2) is done by diagonalisation. It should thus come as no big surprise that this construction needs a few enumerated families. The first convention is to say that "$\{R_i\}$ is an effective enumeration of reductions", which means the following.

Convention 3 $\{R_i\}_{i\in\mathbb{N}}$ *is an enumeration of reduction, and* R *is a transducer such that* $R(i,x)$ *computes* $R_i(x)$ *in* $\mathsf{SPACE} = i\log(n) + i$.

The existence of such an enumeration is proved by using "yardstick" $\mathcal{TM}$'s, and can be found in the classical complexity theory literature, such as [BDG94a] and [BDG94b]. Similarly, we need $\{L_i\}_{i\in\mathbb{N}}$ to be an effective enumeration of PSPACE.

Convention 4 $\{L_i\}_{i\in\mathbb{N}}$ *is an enumeration of* PSPACE, *and* M *is a* $\mathcal{TM}$ *such that* $M(i,\cdot) = M_i(\cdot)$ *decides* L_i *in* $\mathsf{SPACE} = n^i + i$.

We next fix a notation for a/the canonical complete language for PSPACE and its associated reductions.

Convention 5 $\mathcal{C}_{\text{PSPACE}} := \{(M_i, 1^{|x|^i+i}, x) \mid M_i$ *accepts* x *in* $\mathsf{SPACE} = |x|^i + i\ \}$ *is called the "canonial"* PSPACE *complete language.* $\tilde{R}_i : \{0,1\}^* \to \{0,1\}^*$ *defined by* $\tilde{R}_i(x) := (M_i, 1^{|x|^i+i}, x)$ *is called the* i*th canonical reduction, and* $\tilde{R}$ *is a/the transducer such that* $R(i,\cdot)$ *computes* $\tilde{R}_i$ *in* $\mathsf{SPACE} = i\log n + i$.

In the diagonalisation process used in subsection 3.2 to construct an almost complete set for PSPACE, there is another enumeration which will be required. This enumeration is a family of intervals $\{I_i\}$ forming a partition of $\{0,1\}^*$, with the property that the "length" of these intervals grows exponentially fast.

Definition 3.3 *Let* $f : \mathbb{N} \to \mathbb{N}$ *be defined recursively by* $f(0) = 0$ *and* $f(i+1) = 2^i f(i)^i + 1$. $\{I_i\}_{i\in\mathbb{N}}$ *is defined by* $I_i = \{\omega \in \{0,1\}^* | f(i) \leq |\omega| < f(i+1)\}$.

The following remark on the length of words that can be found in a given interval I_i and on the rate of growth of f is trivial, but it is nice to put it down once and for all, since it will appear further in this article at a point where we will too concerned with other difficulties to battle with such details.

Remark 1 *Let $n := |s_N| \simeq \log N$. Let i be the index such that $s_N \in I_i$. The two following facts will be useful: 1) $n < f(i+1)$, which can be rewritten as $N < 2^{f(i+1)}$. 2) $2^i \leq f(i) \leq n$, or alternatively $i \leq \log(f(i)) \leq \log(n)$.*

The diagonalisation process used in the next subsection requires us to distinguish, in each interval I_i, a subset D_i called the distinguished words of I_i. The distinguished words will be used to separate the intervals I_i in four groups, but this will be clear when they are used.

Definition 3.4 *For every $i \in \mathbb{N}$, D_i is defined as the i^2 first words of I_i (under the canonical ordering), and is called the set of distinguished words of I_i.*

A first glimpse of the diagonalisation that is planned to be in the next subsection may be suggested by the following remark, which possibly gives a hint on how the enumerations $\{I_i\}$ and $\{R_i\}$ will be related.

Remark 2 $\forall x \in D_i$, $R_i(x) \in \bigcup_{j \leq i} R_i(x)$.

Notice that this remark is true simply because on input $x \in D_i$, the transducer R_i has not got enough time (using trivial space-time tradeoffs) to output a word of size greater than $f(i+1)$. We point out another trivial fact that we want to free our minds of.

Remark 3 *If x is a distinguished word of I_i, then $|x| = f(i)$.*

Finally, we need a last definition which separates the intervals $\{I_i\}$ in four groups, depending on the value of a function g defined thereunder. This separation into four groups will be used in the diagonalisation process of the next subsection, to define a recursive construction depending on the values of g.

Definition 3.5 *The function $g : \mathbb{N} \to \mathbb{N}$ is defined by $g(i) := 1$ if $\exists x \in D_i$ such that $R_i(x) \notin I_i$, $g(i) := 2$ if $\exists x_1, x_2 \in D_i$ such that $x_1 \neq x_2$ and $R_i(x_1) = R_i(x_2)$, $g(i) := 3$ if $\exists x \in D_i$ such that $R_i(x) \notin \bigcup_{j \leq i} Im(\tilde{R}_j)$ and $g(i) := 4$ otherwise.*

3.2 Construction of an Almost Complete Set

We use a diagonalisation process, which on parameter $\mathcal{C}$ a language, produces another language $A_{\mathcal{C}}$. The process is subtle, since it has to achieve three different tasks at the same time, the two last ones being apparently contradictory: first, it should be that if $\mathcal{C} \in$ PSPACE, then $A_{\mathcal{C}} \in$ PSPACE. Now suppose that $\mathcal{C}$ is complete for PSPACE. The process should insure that $A_{\mathcal{C}}$ is different from $\mathcal{C}$, but similar at the same time! More precisely, if $\mathcal{C}$ is complete, $A_{\mathcal{C}}$ should not be complete, but nevertheless many languages of PSPACE should reduce to it, that is, $A_{\mathcal{C}}$ should be almost complete. The definition of $A_{\mathcal{C}}$ is done with the following idea: at first, we start with the language $A_{\mathcal{C}} := \mathcal{C}$ and the empty language $B_{\mathcal{C}}$. Then we modify $B_{\mathcal{C}}$ and $A_{\mathcal{C}}$ so that $B_{\mathcal{C}} \not\leq A_{\mathcal{C}}$, trying to modify as little as possible the language $A_{\mathcal{C}}$. More precisely, the definition is the following.

Definition 3.6 *Let $\mathcal{C}$ be a language. The languages $A_{\mathcal{C}}$ and $B_{\mathcal{C}}$ are defined recursively on the intervals of the partition I of definition 3.3. On the interval I_0, we put $A_{\mathcal{C}}(x) = \mathcal{C}(x)$ and $B_{\mathcal{C}}(x) = 0$. Suppose that $A_{\mathcal{C}} \cap I_j$ and $B_{\mathcal{C}} \cap I_j$ are defined $\forall j < i$, $A_{\mathcal{C}} \cap I_i$ and $B_{\mathcal{C}} \cap I_i$ are defined according to one of the four following cases, depending on the value of $g(i)$.*

(1) If $g(i) = 1$. *In this case, we leave $A_{\mathcal{C}}$ unchanged, $A_{\mathcal{C}}(x) := \mathcal{C}(x)$ $\forall x \in I_i$. By case assumption, there exists $\xi := \min\{x \in D_i | R_i(x) \notin I_i\}$, and we use this fact to insure that $B_{\mathcal{C}} \not\leq A_{\mathcal{C}}$ via R_i, by setting $B_{\mathcal{C}}(x) := 1 - A_{\mathcal{C}}(R_i(x))$ if $x = \xi$ and $B_{\mathcal{C}}(x) = 0$ if $x \in I_i \setminus \{\xi\}$.*

(2) If $g(i) = 2$. *$A_{\mathcal{C}}$ is left unchanged again, $A_{\mathcal{C}}(x) := \mathcal{C}(x)\ \forall x \in I_i$. Let $\xi_1 < \xi_2$ be the two smallest elements (which exist by case assumption) of D_i such that $R_i(\xi_1) = R_i(\xi_2)$. Once again, we insure that $B_{\mathcal{C}} \not\leq A_{\mathcal{C}}$ via R_i by defining $B_{\mathcal{C}}(x) := 1$ if $x = \xi_1$ and $B_{\mathcal{C}}(x) = 0$ if $x \in I_i \setminus \{\xi_1\}$.*

(3) If $g(i) = 3$. *Let $\xi := \min\{x \in D_i | R_i(x) \notin \bigcup_{j \leq i} \tilde{R}_j\}$. This time we have to modify $A_{\mathcal{C}}$, by setting $A_{\mathcal{C}}(x) := 1$ if $x = R_i(\xi)$, $A_{\mathcal{C}}(x) = \mathcal{C}(x)$ if $x \in I_i \setminus \{R_i(\xi)\}$. We let $B_{\mathcal{C}} \cap I_i$ be the empty language, $B_{\mathcal{C}}(x) := 0\ \forall x \in I_i$, and thus it holds that $B_{\mathcal{C}} \not\leq A_{\mathcal{C}}$ via R_i since $B_{\mathcal{C}}(\xi) \neq A_{\mathcal{C}}(R_i(\xi))$.*

(4) If $g(i) = 4$. *By case assumption, $R_i(D_i) \subseteq \cup_{j \leq i} Im(\tilde{R}_i) \cap I_i$, and $R_{i_{|D_i}}$ is one-to-one, since otherwise $g(i) \leq 2$. Since $|D_i| \stackrel{!}{=} i^2$, there exists $e_i := \min\{1 \leq l \leq i \ : \ |R_i(D_i) \cap Im(\tilde{R}_l)| \geq i\}$. Let J_i be the first i elements of D_i that are mapped by R_i to $Im(\tilde{R}_{e_i})$, and let $F_i := R_i(J_i)$. If $\mathcal{C} \cap F_i = \emptyset$, then let $\xi := \max F_i$. We change $A_{\mathcal{C}}$ by setting $A_{\mathcal{C}}(x) := 1$ if $x = \xi$, $A_{\mathcal{C}}(x) = \mathcal{C}(x)$ if $x \in I_i \setminus \{\xi\}$. $B_{\mathcal{C}}(x) := 0\ \forall x \in I_i$. Notice that if $\mathcal{C} \cap F_i \neq \emptyset$, then trivially $B_{\mathcal{C}} \not\leq A_{\mathcal{C}}$ via R_i, (because $\exists x \in J_i$ such that $A_{\mathcal{C}}(R_i(x)) = \mathcal{C}(R_i(x)) = 1 \neq B_{\mathcal{C}}(x)$), and if $\mathcal{C} \cap F_i \neq \emptyset$, ξ is here to ensure that $B_{\mathcal{C}} \not\leq A_{\mathcal{C}}$ via R_i.*

As emphasised throughout the above definition, $A_{\mathcal{C}}$ and $B_{\mathcal{C}}$ are constructed to ensure that $B_{\mathcal{C}} \not\leq A_{\mathcal{C}}$. The next remark is more precise.

Remark 4 *$\forall i \in \mathbb{N} \exists x \in I_i$ such that $B_{\mathcal{C}}(R_i(x)) \neq A_{\mathcal{C}}(x)$.*

To show the strength of the above construction,. we will start proving some properties of it although, as announced, the main result (which is the fact that $A_{\mathcal{C}}$ is almost complete for PSPACE if $\mathcal{C}$ is chosen wisely), is kept for subsection 3.3. The rest of this subsection is dedicated to the analysis of the space complexity required to decide $A_{\mathcal{C}}$ and $B_{\mathcal{C}}$ (depending on the choice of $\mathcal{C}$). To come to this end, we start by giving a bound on the space required to compute the index of the interval a word is in.

Lemma 3.1 *There exists an algorithm* Index *that runs in* SPACE $= \mathcal{O}(\log n)$ *such that $\forall x \in \{0,1\}^*$,* Index$(x) = i$, *where i is the index such that $x \in I_i$.*

Proof. Let $s_N \in \{0,1\}^n$ be the input to Index. We sketch a very simple algorithm computing Index. *First, compute and store n. Then compute the successive $f(i)$s for $i = 0, 1, 2, 3, ...$ until i is such that $f(i+1) > n$, and output i.* Le us analyse the working space required to run this algorithm. How much space does it take to compute $f(j+1)$ having $f(j)$ and j stored in memory? By definition, $f(j+1) = 2^j f(j)^j + 1$. Using lemma 2.7, we bound the working space required to compute this value by $\mathcal{O}(\log(f(j)))$. Since in our case, $j \leq i$, this is bounded by $\mathcal{O}(\log n)$. Note that we can stop the computation as soon as we see that $f(i+1)$ is going to require more than $\log n$ bits to be stored, and thus we never need to actually store in memory the value $f(i+1)$. □

Since $A_{\mathcal{C}}$ and $B_{\mathcal{C}}$ are defined on each interval, applying a different rule depending on the value of g on this interval, the natural plan to construct an algorithm deciding $A_{\mathcal{C}}$ (or $B_{\mathcal{C}}$) on input x, is to have a subroutine finding the interval such that $x \in I_i$, and another computing the value of g on this interval. The first step of this plan is already realised through the algorithm Index of the previous lemma, the next step is described in the lemma to come.

Lemma 3.2 *There exists an algorithm* Case *that runs in space $\mathcal{O}(\log^3 n)$ such that $\forall x \in \{0,1\}^*$,* Case$(x) = g(i)$, *where i is the index such that $x \in I_i$ and g is the function of definition 3.5.*

Proof. Let x be the input of size n. First use Index to compute and store the index i such that $x \in I_i$. Next, compute and store the value $f(i)$. (It is trivial from the proof of lemma 3.1 that these two computation steps can be carried out within the desired space bounds). Next, compute and store in a list the set D_i, which is of size i^2, and whose elements are of length $f(i)$. The output of this computation cannot be stored within desired space bounds (although the working space of the computation is easily shown to hold within the desired space bounds), since $f(i)$ could be as big as n. To overcome this difficulty, the set D_i has to be stored in a "compressed" form.

The compression takes advantage of the fact that each element of D_i has only $\log i^2$ significant bits. Thus D_i is compressed and stored on $i^2 \log i^2$ bits, which stays within the desired bound, since $i \leq \log n$. (Notice that having the value $f(i)$ stored in memory, "decompression" of D_i, is easy: it consists of padding $f(i) - \log^2(i)$ zeros in front of any compressed object). Next, test whether $g(i) = 1$, that is, test whether $\exists x \in D_i$ such that $|R_i(x)| \leq f(i)$. It is easy to see that this computation can be done in space $\mathcal{O}(i \log n + i)$, which is the space needed to compute R_i on elements of size n, and since the elements of D_i are no longer than x, the input. (Remember that if f and g are space efficiently computable, then so is $g \circ f$, even though the output of f may be too large to be stored, and therefore decompressing the element of D_i to feed them to R_i is not a problem). Next, unless $g(i)$ is already known to be equal to 1 (in which case the algorithm terminates outputting 1), test whether $g(i) = 2$. That is, test whether $\exists x_1 \neq x_2 \in D_i$ such that $R_i(x_1) = R_i(x_2)$. The computation of $R_i(x)$ with $x \in D_i$ is bounded as for the case where $g(i) = 1$. This time, the computation of $R_i(x_1)$ and $R_i(x_2)$ for (x_1, x_2) ranging in $D_i \times D_i$ have to be done in parallel, comparing the output of both computations bitwise each time a bit is outputed by one of the computations. This lets the space complexity of this computation step be bounded by $\mathcal{O}(i \log n + i)$. Next, unless $g(i)$s value is already known, test whether $g(i) = 3$. That is, test whether there exists $y \in D_i$ such $R_i(y) \in \bigcup_{j \leq i} Im(\tilde{R}_j)$. To be convinced that this step can be carried out space efficiently, let us look at one of the tests that have to be made. (All of which can be carried out one at a time, after having cleared the memory space previously allocated). Let $j \leq i$ and $y \in D_i$ be fixed, and suppose it needs to be decided whether $R_i(y) \in Im(\tilde{R}_j)$. That is, it has to be decided whether $R_i(y)$ is of the form $(M_j, 1^{|z|^j + j}, z)$ for a certain $z \in \{0,1\}^*$ and a certain $j \leq i$. The only difficult thing to see is that the number of 1s is correct. To verify this, we need to compute $|z|^j + j$, and compare it to the number of 1's. This will take $\mathsf{SPACE} = \log(|z|^j + j)$, which is adequate, since we can bound $|z|^j + j$ by $2^{\mathcal{O}(\log^3 n)}$, noticing that since $y \in D_i$, it holds that $|y| \leq n$, and thus (through trivial space-time tradeoffs), the length of the output of $R_i(y)$, and thus of z, is bounded by $2^{\mathcal{O}(i \log n)}$. This gives an upper bound of $2^{\mathcal{O}(\log^2 n)}$ on the length of z and thus $|z|^j + j \leq |z|^i + i \leq |z|^{\log n} + \log n \leq 2^{\mathcal{O}(\log^3 n)}$, as announced. Finally, if none of the previous tests where successful, then $g(i) = 4$. □

To finish the analysis of the space complexity required to compute $A_{\mathcal{C}}$ and $B_{\mathcal{C}}$, we only need to take advantage of the two previous lemmas, and optimise the computations naturally arising from definition 3.6 while trying to decide $A_{\mathcal{C}}$ and $B_{\mathcal{C}}$.

Theorem 3.3 *Let $t(n)$ be a complexity function such that $\mathcal{C}$ is in $\mathsf{SPACE} = t(n)$. The languages $A_{\mathcal{C}}$ and $B_{\mathcal{C}}$ are in $\mathsf{SPACE} = \max\{t(n), \log^3(n)\}$.*

Proof. We exhibit an algorithm deciding $A_{\mathcal{C}}$ within the announced space bound. An algorithm for $B_{\mathcal{C}}$ can be obtained similarly. Let $x \in \{0,1\}^n$ be the input. First compute, in space $\mathcal{O}(\log^3 n)$, i and $g(i)$ such that $x \in I_i$, using the algorithms `Index` and `Case` of lemmas 3.1 and 3.2. If $g(i) = 1$ or $g(i) = 2$, then output $\mathcal{C}(x)$, which is computed in space $t(n)$. Otherwise if $g(i) = 3$, find ξ, the smallest element of D_i such that $R_i(\xi) \notin \bigcup_{j \leq i} Im(\tilde{R}_j)$. In order to find ξ, it has to be tested (for each $x \in D_i$) whether $R_i(x) = (M_j, 1^{|y|^j + j}, y)$, for a certain $j \leq i$ and for a certain $y \in \{0,1\}^*$. As argued in the proof of lemma 3.2, these tests can be carried out one at a time in $\mathsf{SPACE} = \mathcal{O}(\log^3 n)$. Next, compute $R_i(\xi)$, and output 1 if $x = R_i(\xi)$, output $\mathcal{C}(x)$ otherwise. Computing $R_i(\xi)$ takes $\mathsf{SPACE} = i \log(|\xi|) + i$, which is $\mathcal{O}(\log^2 n)$ since $i \leq \log n$ and $\xi \in D_i \Rightarrow |\xi| \leq |x| = n$, (and of course, computing $\mathcal{C}(x)$, if required, takes $\mathsf{SPACE} = t(n)$). If $g(i) = 4$, then compute $e_i := \min\{l \in \{1, ..., i\} : |R_i(D_i) \cap Im(\tilde{R}_l)| \geq i\}$. This computation only needs to do simple counting of how many elements of D_i are mapped to $Im(\tilde{R}_j)$ by R_i, for each $j \leq i$ and for x's ranging over D_i. Once again, the same difficulty arises: test whether $R_i(x) \in Im(\tilde{R}_j)$ for a certain $j \leq i$, which we have already shown to be computable in $\mathsf{SPACE} = \mathcal{O}(\log^3 n)$. Once e_i is found, compute and store a compressed list of J_i, the first i^2 elements of D_i being mapped to $Im(\tilde{R}_{e_i})$ by R_i, and a compressed list of $F_i := R_i(J_i)$, (taking advantage, as in the proof of lemma 3.2, of the fact that elements of D_i have at most i^2 significant bits). The compressed list of J_i and F_i are in fact the same, only the decompression algorithm changes, c.f. below. Then, find $\bar{y}$ the compressed representant of $y \in J_i$ such that $R_i(y) =: \xi$ is $\max F_i$. This can be done

by decompressing and computing R_i on every element of J_i, and comparing them pairwise and bitwise, (in order to avoid having to store the decompressed elements in memory). Once again, a difficulty arises from the fact that y is too large to be stored in memory. Thankfully, $\bar{y}$ is not. The values that have to be computed are $R_i(y) = R_i \circ \texttt{Decompression}(\bar{y})$, (where `Decompression` only consists of padding $f(i) - i$ 0's before the compressed element), for $y \in J_i$. Since R_i, and the decompression algorithm both run in $\mathsf{SPACE} = i \log n + i = \mathcal{O}(\log^2 n)$, this bounds the working space required for this computation step.If $x \neq \xi$, output $C(x)$.Otherwise, (if $x = \xi$), we have to check whether $\mathcal{C} \cap F_i = \emptyset$, and output 1 if this is the case, and output $\mathcal{C}(x)$ otherwise. To test whether $\mathcal{C} \cap F_i = \emptyset$ we have to to compute $\mathcal{C}(y)$ $\forall y \in F_i$. This is possible in $\mathsf{SPACE} = t(n)$, since $x = \xi = maxF_i \Rightarrow |y| \leq n \ \forall y \in F_i$. A similar proof shows that the same space bounds hold to compute $B_{\mathcal{C}}$. □

3.3 Proving the Almost Completeness of $A_{\mathcal{C}_{\texttt{PSPACE}}}$

The main result of this article is the fact that, when one plugs in $\mathcal{C}_{\texttt{PSPACE}}$ as a parameter in the construction of the previous section, one obtains an almost complete set for PSPACE.

Theorem 3.4 *$A_{\mathcal{C}_{\texttt{PSPACE}}}$ is almost complete (for* PSPACE*).*

Let us comment on the structure we want to give to this section. We find that it is easier to follow the guiding line if we state (and prove) results by decreasing order of importance, leaving the technical results for the end. Following this idea, we started this section by stating theorem 3.4, and we shall also prove it immediately. Of course, the disadvantage is that the proof relies results that are not yet proved, and that the reader must admit temporarily. The proof of theorem 3.4 follows.

Proof. Theorem 3.3 implies that $A_{\mathcal{C}_{\texttt{PSPACE}}} \in \texttt{PSPACE}$, and that $B_{\mathcal{C}_{\texttt{PSPACE}}} \in \texttt{PSPACE}$. Since by construction $B_{\mathcal{C}_{\texttt{PSPACE}}} \not\leq A_{\mathcal{C}_{\texttt{PSPACE}}}$, then $A_{\mathcal{C}_{\texttt{PSPACE}}}$ is a language of PSPACE which is not complete. We would be finished if we could prove that $\mu(P_m(A_{\mathcal{C}_{\texttt{PSPACE}}})) = 1$. We prove this later fact in lemma 3.8. □

It is now time to make use of definition 3.6. In this definition, the function g separating the intervals $\{I_i\}_{i\in\mathbb{N}}$ in four cases was defined. Recall that for i's where $g(i) = 4$, sets J_i and F_i were defined. These sets will be used to define a family of martingales which we need to prove our main result.

Convention 6 *$\forall i \in \mathbb{N}$ such that $g(i) = 4$, $e_i \in \mathbb{N}$ and the sets J_i and F_i are defined as in the case 4 of definition 3.6. If $g(i) \neq 4$, we do not define e_i nor J_i, and we let $F_i := \emptyset$.*

Roughly speaking, we are going to finish showing that $A_{\mathcal{C}_{\texttt{PSPACE}}}$ is almost complete (i.e. prove lemma 3.8) by showing that the languages of PSPACE that do not reduce to it are rare, i.e. of null measure. To prove this fact, we will show that it is possible to cover those languages with a (family of) $\Gamma(\texttt{PSPACE})$ martingale(s). Describing these martingales is the next job to be completed, and we shall start now. These martingales will have a strategy (in reference to the "casino game"), which is to always make the assumption that the languages that do not reduce to PSPACE do not contain many words in a given family of sets $\{W_i\}_{i\in\mathbb{N}}$ defined below, using the two sets of convention 6. The detailed description of these martingale, and the reason why this strategy brings us to our goals will arrive soon. First we define the family of W_i's mentioned above.

Definition 3.7 *For all $e, i \in \mathbb{N}$, $W_{i,e}$ is defined as $\tilde{R}_e^{-1}(F_i)$. $W'_{i,e}(x)$ is the set of words of $W_{i,e}$ that are smaller than x, under the canonical ordering.*

As explained above, to prove lemma 3.8 we need to prove that the languages that do not reduce to $A_{\mathcal{C}_{\texttt{PSPACE}}}$ are a measure null set. This is done by showing that there is a (suitable family of) martingale(s) covering this set. Since the formal definition 3.8 of these martingales is not very intuitive, we start by giving an informal definition/explanation of these martingales, in the form of a family of "betting strategies" to play the "casino game". (c.f. [ASMRT00] for a nice

description of the casino game, and of betting strategies). We describe informally a family of martingales/betting strategies $\{d_e\}_{e\in\mathbb{N}}$, such that d_e wagers on words of $\cup_{i\in\mathbb{N}}W_{i,e}$ only. Fix $e \in \mathbb{N}$, and let us describe d_e. Consider the following martingale while playing the *casino game*: first split the initial capital $c = 1$ into sub-capitals $c_i = \frac{1}{2^i}$, for $i \in \mathbb{N}^*$. The ith capital will be used to bet on words of $W_{i,e}$ only, according to the following strategy. Suppose the martingale is playing against a language L, i.e. for the martingale to win against (or cover) the language, it must hold that $\limsup_{N\to\infty} d_e(\chi_L[N]) = \infty$. When it comes to bet on a word s_N, the martingale first looks whether $s_N \in W_{i,e}$ for some $i \in \mathbb{N}$. If this is not the case, then no money is wagered (i.e. $d(\chi[1, N-1]1) = d(\chi[1, N-1]0) = d(\chi[1, N-1])$). If $s_N \in W_i$ for some $i \in \mathbb{N}$, the strategy wagers *all the current value of capital* c_i on the fact that the word s_N is *not* in L. If this is the case, the current capital c_i is doubled, otherwise it is completely lost, and thus the current value of c_i becomes null. Simple, is it not? The hard bit will be to show how to compute this family of martingales, and to show that they cover any language (of PSPACE) that does not reduce to $A_{\mathcal{C}_{\text{PSPACE}}}$. (More precisely, we will show that if $L_e \not\leq A_{\mathcal{C}_{\text{PSPACE}}}$, then $L_e \in S^\infty[d_e]$). Let us give a hint on the way this is done, by showing a sufficient condition for a language to be covered by d_e. Since the size of W_i is i whenever $g(i) = 4$, the martingale uses the ith capital $c_i = \frac{1}{2^i}$ to wager exactly i times, (on words of $W_{i,e}$), and since the strategy always plays the total amount of the current value of c_i, sooner or later, c_i is either doubled i times, reaching the peak value of 1 (this happens if $L \cap W_{i,e} = \emptyset$), or it is completely lost, (if $L \cap W_{i,e} \neq \emptyset$). If a language L is such that there are infinitely many i's such that $L \cap W_{i,e} = \emptyset$, then there are infinitely many sub-capitals which reach the peak value of 1, and thus the total current capital $c = \sum_{i\in\mathbb{N}^*} c_i$ tends to infinity as the casino game goes on, (i.e. $\limsup_{N\to\infty} d(\chi[1, N]) = \infty$) and L is covered by the strategy. Thus if any language L_e of PSPACE that does not reduce to $A_{\mathcal{C}_{\text{PSPACE}}}$ satisfies $L_e \cap W_{i,e} = \emptyset$ for infinitely many i's, we are close to having proved what we wanted. We are now going to state things more rigorously, and to prove step by step the points argued above. First of all, we define rigorously the martingales d_e.

Definition 3.8 *Let L be a language and $e \in \mathbb{N}$. For any $N \in \mathbb{N}$, the function d_e is defined by $d_e(\chi_L[0, N]) := \sum_{j\in\mathbb{N}^*} c_j(\chi_L[0, N])$, where the "sub capitals" $c_j(\chi_L[0, N])$ are defined by induction on N. At the first step of the recursion ($N = 0$), the $c_j(\chi_L[0, 0])$'s are initialised to $c_j(\chi_L[0, 0]) := \frac{1}{2^j}$. At the Nth step of the recursion, the values of the c_j's are updated according to the following rule: let i be the index such that $\tilde{R}_e(s_N) \in I_i$. If $j \neq i$, c_j is left unchanged. If $j = i$, but $s_N \notin W_{i,e}$, c_j is left unchanged too. If $j = i$ and $s_N \in W_{i,e}$, then $c_j = c_i$ is modified according to the following rule: $c_i(\chi_L[0, N]) := 2c_i(\chi_L[0, N-1])$ if $s_N \notin L$ (i.e if $\chi_L[N] = 0$), and $c_i(\chi_L[0, N]) := 0$ otherwise, (i.e. if $\chi_L[N] = 1$).*

Convention 7 *Whenever it is clear from the context what is the language L referred to, we replace the notation $c(\chi_L[0, N])$ by $c(N)$.*

From the definition above, it should be clear that each function d_e is a martingale. The reader already convinced of it can easily skip the next lemma, that we include for completeness.

Lemma 3.5 *For any $e \in \mathbb{N}$, d_e is a martingale.*

Proof. We have to verify that (for any e) $d_e : \{0,1\}^* \to \mathbb{R}^+$, and that it satisfies the equality of definition 2.3. By definition of the c_j's, it holds that $\sum_{j=1}^{\infty} c_j(0) = 1$. Also, there are at most N sub capitals c_j such that $c_j(N) \neq c_j(0)$, and these sub capitals satisfy $0 \leq c_j(N) \leq 1$. Therefore, since (for any L) $d_e(\chi_L[0, N]) = \sum_{i\in\mathbb{N}^*} c_i(N)$, it holds that $0 \leq d_e(\chi_L[0, N]) \leq N + 1 \in \mathbb{R}^+$. Next, we need to check that $d_e(\chi_L[0, N-1]) = \frac{d_e(\chi_L[0,N-1]1)+d_e(\chi_L[0,N-1]0)}{2}$. Two cases have to be distinguished: either $\forall j \in \mathbb{N}^*$, $c_j(N) = c_j(N-1)$, and thus the equality is trivial, either one (and only one) of the capitals $c_j(N)$ changes at the Nth step of the computation. If this is so, the equality still holds, since $c_j(\chi_L[0, N-1]1) = 0$ and $c_j(\chi_L[0, N-1]0) = 2c_j(\chi_L[0, N-1]) = 0$. □

Remember that from convention 4, we have $\{L_i\}_{i\in\mathbb{N}}$ an enumeration of PSPACE. The next lemma is quite important, and it says that the eth martingale d_e has the property of covering (the

eth language) L_e, if it does not reduce to $A_{\mathcal{C}_{\texttt{PSPACE}}}$. This finishes demistifying the role of the d_e's. The proof is simple, if we have a technical result (lemma 3.7), but following the plan exposed at the beginning of this subsection, we prove this technical result separately.

Lemma 3.6 *For any $e \in \mathbb{N}$, if L_e does not reduce to $A_{\mathcal{C}_{\texttt{PSPACE}}}$, then $L_e \in S^\infty[d_e]$.*

Proof. This lemma is proved using the next one, (lemma 3.7), which states that if $L_e \not\leq A_{\mathcal{C}_{\texttt{PSPACE}}}$, then $W_{i,e} \cap L_e = \emptyset$ for infinitely many i's in $\mathbb{N}$. This implies that, while playing the casino game versus the language L_e, the martingale d_e sees infinitely many of its sub-capitals c_i reaching the peak value of 1, which in turn implies that $\limsup_{N\to\infty} d_e(\chi_L[0,N]) = \infty$. □

The technical lemma required to finish the proof above is the following.

Lemma 3.7 ([ASMRT00]) *If L_e does not reduce to $A_{\mathcal{C}_{\texttt{PSPACE}}}$, then there are infinitely many i's in $\mathbb{N}$ such that $W_{i,e} \cap L_e = \emptyset$.*

The fact that this lemma holds comes from the structure induced by the diagonalisation process (of definition 3.6), and its proof can be found in full details in [ASMRT00]. Nevertheless, we give the main ideas of the proof.

Proof. Suppose that $L_e \not\leq$ to $A_{\mathcal{C}_{\texttt{PSPACE}}}$. Then necessarily, it does not reduce to $A_{\mathcal{C}_{\texttt{PSPACE}}}$ via a finite variation of $\tilde{R}_e$. This implies that there are infinitely many words x such that $A_{\mathcal{C}_{\texttt{PSPACE}}}(\tilde{R}_e(x)) \neq \mathcal{C}_{\texttt{PSPACE}}(\tilde{R}_e(x))$, since L_e reduces to $\mathcal{C}_{\texttt{PSPACE}}$ (via $\tilde{R}_e$), and not to $A_{\mathcal{C}_{\texttt{PSPACE}}}$. By analysing the construction of $A_{\mathcal{C}_{\texttt{PSPACE}}}$ (definition 3.6), we see that this implies that $[g(i) = 4$ and $e_i = e$ and $\mathcal{C} \cap F_i = \emptyset]$ for infinitely many i's in $\mathbb{N}$, (detailed explanation of this latter fact can be found in [ASMRT00]). We then notice that for those infinitely many i's, the following holds. $\mathcal{C} \cap F_i = \emptyset \Rightarrow \tilde{R}_{e_i}^{-1}(\mathcal{C}) \cap \tilde{R}_{e_i}^{-1}(F_i) = \emptyset \Rightarrow L_e \cap W_{i,e} = \emptyset$, which finishes the proof. (c.f. figure 1 for an illustration of the "state of the world" for those cases). □

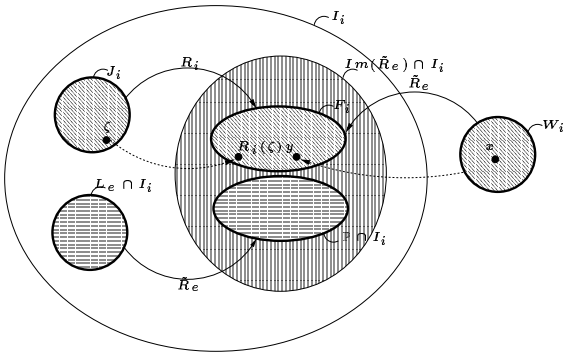

Figure 1: Cases where the $c_i(N) = 1$ for $N >>$

We can finally state the lemma we need to finish the proof of our main result: theorem 3.4.

Lemma 3.8 *The set $\mathcal{L} := \{L \in \texttt{PSPACE} \mid L \not\leq A_{\mathcal{C}_{\texttt{PSPACE}}}\}$ is a measure null set: $\mu(\mathcal{L}) = 0$.*

Once again, we can prove this lemma very easily, except for a technical result that we prove separately in lemma 3.9.

Proof. Lemma 3.6 implies that $\mathcal{L} \subseteq \cup_{e\in\mathbb{N}} S^\infty[d_e]$, thus we will be finished if we manage to prove that $\mu(\cup_{e\in\mathbb{N}} S^\infty[d_e]) = 0$. We prove this latter fact in lemma 3.9. □

We only need a last technical lemma to complete the demonstartion.. This last technical lemma is also the main technical result of this section. It states that a given union of sets is of null measure. We know that this holds when the union is an "easy union" of null sets, c.f. the first measure axiom, in section 2.3, and this is the way the proof is obtained.

Lemma 3.9 $\mu(\cup_{e\in\mathbb{N}} S^\infty[d_e]) = 0$

Proof. The next claim implies that $\cup S^{\infty}[d_e]$ is an easy union of null sets. The invocation of lemma 2.12 then terminates the proof.

Claim 1 *There exists T_N a transducer such that: 1) $T_N(e,x) = d_e(x)$. 2) $\exists k \in \mathbb{N}$ such that $\forall e \in \mathbb{N}$, $T_N(e,\cdot)$ computes in $\mathsf{SPACE} = (e^k + k)(\log^k(N) + k)$, querying (the right hand side of) its input in a dependency set $\{G_{e,N}\}_{N\in\mathbb{N}}$ such that $|G_{e,N}| \leq (e^k + k)(\log^k(N) + k)$.*

Let us substantiate this claim. Let $i = i_N$ be the index such that $\tilde{R}_e(s_N) \in I_i$. Notice that by definition of the c_j's, and since $\tilde{R}_e$ is increasing (with respect to the canonical ordering), it holds that $\forall j > i$, $c_j(N) = c_j(0)$. We can thus rewrite $d_e(\chi_L[0,N])$ in a more convenient way: $d_e(\chi_L[0,N]) \stackrel{!}{=} \sum_{j=1}^{\infty} c_j(N) = \sum_{j>i} c_j(0) + \sum_{j=1}^{i} c_j(N) = \sum_{j>i} \frac{1}{2^j} + \sum_{j=1}^{i} c_j(N) = \frac{1}{2^i} + \sum_{j<i} \frac{2^{|W_{j,e}|}}{2^j}\delta(W_{j,e}) + \frac{2^{|W'_{i,e}(s_N)|}}{2^i}\delta(W'_{i,e}(s_N))$, where $\delta(A) = 1$ if $A \cap L = \emptyset$, and $\delta(A) = 0$ otherwise. We are going to show that this sum can be computed as efficiently as stated in claim 1. First, we shall give an algorithm computing the last term of the sum above, whose analysis can be summarised in the next claim:

Claim 2 *Let $n := \log N$. $\exists T'_N$ a transducer such that $\forall e \in \mathbb{N}$, $T'_N(e, \chi_L[0,N])$ computes $\frac{2^{|W_{j,e}|}}{2^j}\delta(W_{j,e})$ in $\mathsf{SPACE} = \mathcal{O}(e^3 \log^3 n)$ and querying the right-hand side of its input in a dependency set of size $\mathcal{O}(e^2 \log^2 n)$.*

Before we substantiate this claim, let us discuss its consequences, in particular, the fact that it implies that claim 1 holds (and thus that the proof of the lemma is finished). As will be seen below, claim 2 is substantiated by the exhibition of an algorithm computing the last term of the sum $\frac{1}{2^i} + \sum_{j<i} \frac{2^{|W_{j,e}|}}{2^j}\delta(W_{j,e}) + \frac{2^{|W'_{i,e}(s_N)|}}{2^i}\delta(W'_{i,e}(s_N))$. This algorithm can be easily modified in order to compute any other term of this sum. We thus have implicitly a (uniform) family of algorithms computing the terms of the sum above. What about an algorithm computing the sum itself? Lemma 2.7 permits to bound the resources needed to compute the whole sum, and thus d_e, which substantiates claim 1, as announced. In fact, lemma 2.7 and claim 2, yield an even stronger version of claim 2. Although claim1 is sufficient to finish the proof, we cannot resist giving this stronger version of claim 1 [1].

Claim 3 *Let $n := \log N$. There exists T''_N a transducer such that: 1) $T''_N(e,x) = d_e(x)$. 2) $\exists k \in \mathbb{N}$ such that $\forall e \in \mathbb{N}$, $T''_N(e,\cdot)$ computes in $\mathsf{SPACE} = \mathcal{O}(e^3 \log^3 n)$, querying (the right hand side of) its input in a dependency set $\{G_{e,N}\}_{N\in\mathbb{N}}$ such that $|G_{e,N}| = \mathcal{O}(e^2 \log^2 n)$.*

We still have to substantiate claim 2. We do this by giving an algorithm for the transducer T'_N, (as explained above, it suffices to exhibit an algorithm computing the term $\frac{2^{|W'_{i,e}(s_N)|}}{2^i}\delta(W'_{i,e}(s_N))$).

(1) First, construct and store s_N, querying the auxiliary input N. Let $n := |s_N| \simeq \log N$.

(2) Compute $z := \tilde{R}_e(s_N)$. This takes $\mathsf{SPACE} = n^e + e$, (c.f. convention 5). Notice that z should not be stored, but it should rather be recomputed and accessed bitwise each time it needs to be accessed.

(3) Compute i such that $z \in I_i$. Notice that $|z| = |(M_e, 1^{n^e+e}, s_N)| \leq 3(n^e + e)$. Using the algorithm `Index` of lemma 3.1 as a subroutine, this computation is carried out in $\mathsf{SPACE} = \mathcal{O}(\log|z|) = \mathcal{O}(e \log n)$.

(4) Compute $g(i)$. (This takes $\mathsf{SPACE} = \mathcal{O}(\log^3|z|) = \mathcal{O}(e^3 \log^3 n)$, using algorithm `Case` of lemma 3.2). If $g(i) \neq 4$, then terminate and output the value $\frac{1}{2^i}$, (which only requires $\mathsf{SPACE} = \log|z|$, using remark 1, and since $z \in I_i$).

(5) If $g(i) = 4$, we need to compute $W'_{i,e}(s_N)$, and output $\frac{2^{|W'_{i,e}(s_N)|}}{2^i}$ if $W'_{i,e}(s_N) \cap L = \emptyset$, and output 0 otherwise. This is how it is done:

[1] Although this stronger version of claim 1 is of no use in this proof, it could have implications for further results, as explained in section 4.

(5.1) Compute $e_i := \min\{1 \leq l \leq i : |R_i(D_i) \cap Im(\tilde{R}_l)| \geq i\}$. From (the proofs of) lemmas 3.2 and 3.3, we recall that this can be computed in $\log^3 |z| = \mathcal{O}(e^3 \log^3 n)$.

(5.2) If $e_i \neq e$, then $W_{i,e} = \emptyset$, since $W_{i,e} = \tilde{R}_e^{-1}(R_i(J_i))$, and $R_i(J_i) \subset Im(\tilde{R}_{e_i})$, which has an empty intersection with the co-domain of $\tilde{R}_e$. Thus we terminate by outputting $\frac{1}{2^i}$, (the ith capital is never used to wager, thus it stays at its initial value).

(5.3) If $e_i = e$, we compute $LIST := W'_{i,e}(S_N)$. $LIST$ contains $i \leq \log(|z|)$ elements, all of which are smaller in size than $|z|$. It could thus be stored using space equal to $|z| \log |z|$, but since $W_{i,e} = \tilde{R}_e^{-1}(R_i(J_i))$, we can reduce the space required by storing $LIST$ in a compressed way: we store J_i, which in turn can be compressed, remembering that the elements of J_i only have $\log(i^2) \leq \mathcal{O}(log \log(|z|)) \leq \mathcal{O}(\log\log(n^e)) \leq \mathcal{O}(\log(e \log n))$ significant bits. (c.f. the proof of lemma 3.2). The decompression algorithm can be computed easily, since if $\gamma(\zeta)$ is the "compressed" representative of $\zeta \in W_{i,e}$, the decompression algorithm consists of computing $\tilde{R}_e^{-1} \circ R_i \circ PADD(\gamma(\zeta))$, where $PADD(\gamma(\zeta))$ pads $|z| - \log |i^2|$ zeros to the right of $\gamma(\zeta)$. The decompression algorithm runs in $\mathsf{SPACE} = \mathcal{O}(\log^2 |\zeta|) \leq \mathcal{O}(\log^2 |z|)$, (which is the space required to compute $\tilde{R}_e^{-1} \circ R_i(\zeta)$). Finally, the value to output is: $\frac{2^{|W'_{i,e}|}}{2^i}$ if $W'_{i,e} \cap L = \emptyset$, and 0 otherwise. It is easy to see that the value $\frac{2^{|W'_{i,e}|}}{2^i}$ can be computed in $\mathsf{SPACE} = \log |z|$. What about the condition$W'_{i,e} \cap L = \emptyset$? This is very easily checked, by querying the input, and verifying that $\chi_L[M] = 0$, for every M such that $s_M \in W'_{i,e}$. Since this computation step is the only one requiring to query the input, it permits to bound the size of the set of queried words by $|W_{i,e}| \leq i \leq \log n$. Therefore, on input s_N, the bits queried are contained in $Q_N := \cup_{j \leq i} W_{i,e}$, where i is such that $z \in I_i$, and $i \leq |z| = \mathcal{O}(e \log n)$. The Q_N's form a dependency set of size at most $i^2 = \mathcal{O}(e^2 \log^2 n)$ (since $|W_{i,e}| \leq i$).

□

4 Conclusion

The concept of almost completeness for a complexity class $\mathcal{C}$ defines problems that capture "some of the hardness" of $\mathcal{C}$, without being the hardest ones. It is known that such problems of intermediate hardness exist for big complexity classes, (for which E is a prototype), but it was unknown whether such problems existed for small complexity classes: complexity classes which do not, or are not known to contain E. We answer this question by constructively proving that an almost complete problem exists for PSPACE. Far from closing the chapter, this result rather opens the question of the quantitative-completeness structure of PSPACE, and of other small complexity classes. A sample of the questions that arise: Are there many almost complete problems in PSPACE? i.e. what is the measure of the set of almost complete problems? A language A is almost complete if its lower span $P_m(A)$ is of measure 1, i.e. if $\mu(P_m(A)) = 1$, (and if it is not complete). A is weakly complete if its lower span is of non-null measure, i.e. if $\mu(P_m(A)) \neq 0$. Do weakly complete problems (which are not almost complete) exist for PSPACE? etc...

The most challenging problem is perhaps to investigate the quantitative-completeness structure of P, which is arguably the class of highest interest, as its "nickname" of class of "efficiently decidable problems" perhaps suggests. While class P itself may be very difficult to tackle, there is reasonable hope that the structure of slightly larger classes, such as QP or SUBEXP, could be investigated using the same approach as in this article. The reason for our optimism on the feasibility of this goal is the very high efficiency of the family of martingales of lemma 3.9, which can be computed logarithmically better then what is required, as pointed out in the discussion preceding claim 3. This leaves some room for manoeuvre that could perhaps be used successfully to extend the results of this article to small time complexity classes.

References

[AS94] E. Allender and M. Strauss. Measure on small complexity classes, with applications for BPP. In *Proceedings of the 35th IEEE Annual Symposium on Foundations of Computer Science*, volume 35, pages 807–818, 1994.

[AS95] E. Allender and M. Strauss. Measure on P: Robustness of the notion. In *Proceedings of the 20th Mathematical Foundations of Computer Science*, volume 969, pages 129–138. Springer, 1995.

[AS00] K. Ambos-Spies. Measure theoretic completeness notions for the exponential time classes. In *Mathematical Foundations of Computer Science*, volume 1893 of *Lecture Notes in Computer Science*, pages 152–161. Springer, 2000.

[ASLM98] K. Ambos-Spies, S. Lempp, and G. Mainhardt. Randomness vs. completeness: on the diagonalisation strength of resource bounded random sets. In *Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*, pages 465–473. Springer, 1998.

[ASMRT00] K. Ambos-Spies, W. Merkle, J. Reimann, and S. Terwijn. Almost complete sets. *Preliminary version in Symposium on Theoretical Aspects of Computer Science*, 1770:419–430, 2000. To appear in Theoretical Computer Science.

[ASMZ96] K. Ambos-Spies, E. Mayordomo, and X. Zheng. A comparison of weak completeness notions. In *Proceedings of the 11th Annual Comference on Computational Complexity*, pages 171–178. IEEE Computer Society Press, 1996.

[ASTZ97] K. Ambos-Spies, S.A. Terwijn, and X. Zheng. Resource bounded randomness and weakly complete problems. *Theoretical Computer Science*, 168:195–207, 1997.

[BC94] D.P. Bovet and P. Crescenzi. *Introduction to the Theory of Complexity.* Addison-Wesley, 1994.

[BDG94a] J.L Balcázar, J. Díaz, and J. Gabaró. *Structural Complexity I.* Springer-Verlag, 1994.

[BDG94b] J.L Balcázar, J. Díaz, and J. Gabaró. *Structural Complexity II.* Springer-Verlag 1990, 1994.

[Bus87] S.R. Buss. The boolean formula value problem is in ALOGTIME. In *Symposium on the Theory of Computing*, number 19, pages 123–131. ACM, 1987.

[CSS97] J.-Y. Cai, D. Sivakumar, and M. Strauss. Constant depth circuits and the Lutz hypothesis. *IEEE Symposium on Foundations of Computer Science*, 1997.

[JL95a] D.W. Juedes and J.H. Lutz. The complexity and distribution of hard problems. *SIAM Journal on Computing*, 24(2):279–295, 1995.

[JL95b] D.W. Juedes and J.H. Lutz. Weak completeness in e and e_2. *Theoretical Computer Science*, 143:149–158, 1995.

[Jue95] D.W. Juedes. Weakly complete problems are not rare. *Computational Complexity*, 5:267–283, 1995.

[LM94] J.H. Lutz and E. Mayordomo. Measure, stochasticity, and the density of hard languages. *SIAM Journal on Computing*, 23:762–779, 1994.

[LM96] J.H. Lutz and E. Mayordomo. Cook versus Karp-Levin: Separating completeness notions if NP is not small. *Theoretical Computer Science*, 164(1–2):141–163, 1996.

[Lut92] J.H. Lutz. Almost everywhere high non-uniform complexity. *Journal of Computer and System Science*, 44:220–258, 1992.

[Lut95] J.H. Lutz. Weakly hard problems. *SIAM Journal on Computing*, 24:1170–1189, 1995.

[Lut96] J. H. Lutz. Observations on measure and lowness for Δ_2^{P}. In *Proceedings of the 13th Symposium on Theoretical Aspects of Computer Science*, volume 1046 of *Lecture Notes in Computer Science*, pages 87–97, Berlin, 1996. Springer Verlag.

[Lut97] J. H. Lutz. The quantitative structure of exponential time. In L. A. Hemaspaandra and A. L. Selman, editors, *Complexity Retrospective II*, pages 225–260. Springer, 1997.

[May94a] E. Mayordomo. Almost every set in exponential time is P-bi-immune. *Theoretical Computer Science*, 136:487–156, 1994.

[May94b] E. Mayordomo. *Contribution to the Study of Resource Bounded Measure.* PhD thesis, Universitat Politécnica de Catalunya, Barcelona, 1994.

[Mer95] W. Merkle. Personal communication, 1995.

[Mos02] P. Moser. A generalization of Lutz's measure to probabilistic classes. Technical Report 02-058, Electronic Colloquium on Computational Complexity, October 2002.

[MV93] M.L. and P. Vitànyi. *An Introduction to Kolmogorov Complexity and its Applications.* Springer-Verlag New York, 1993.

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

[Pow02] O. Powell. Measure on P revisited. Technical Report TR02-065, Electronic Colloquium on Computational Complexity, December 2002.

[RS98] K.W. Regan and D. Sivakumar. Probabilistic martingales and BPTIME classes. *IEEE*, pages 186–200, 1998.

[Str97] M. Strauss. Measure on P: Strength of the notion. *Information and Computation*, 136(1):1–23, 1997.

ECCC ISSN 1433-8092
http://www.eccc.uni-trier.de/eccc
ftp://ftp.eccc.uni-trier.de/pub/eccc
ftpmail@ftp.eccc.uni-trier.de, subject 'help eccc'