Two Simple Intrinsic Characterizations of Main Complexity Classes

Paul J. Voda

Institute of Informatics, Comenius University Bratislava,
Mlynská dolina, 842 15 Bratislava, Slovakia,
email: voda@fmph.uniba.sk home-page: www.fmph.uniba.sk/∼voda

Abstract. We give two simple uniform characterizations of the complexity classes $L \subseteq NL \subseteq P \subseteq NP$ by register machine programs. Both characterizations are intrinsic because they do not refer to any time and space bounds. The first characterization, which permits programs to peek into their computation history, captures also *Pspace* and it is completely syntactical in that the restrictions on programs are decided from their syntax. The programs computing predicates of larger classes are permitted to access larger parts of their history. The second characterization restricts the access of programs to parts of their input which are then projected away by existential quantification.

1 Introduction

We present two uniform characterizations by simple register machines of the four unseparated complexity classes $L \subseteq NL \subseteq P \subseteq NP$. The first characterization is by classifying programs by the level of access to their computation history. This characterization also includes the class *Pspace*. The second characterization is by existential projection where we classify the programs by the level of access to a quantified part of the argument. Both characterizations are *intrinsic* in the sense that they do not explicitly refer to any time and space bounds. The first characterization is also syntactic as it classifies programs by easily decidable syntactic criteria.

Intrinsic characterizations are important because they stress the combinatorial nature of the separation problem as opposed to probably much harder number theoretical nature suggested by explicit bounds. The known intrinsic characterizations of computational complexity classes are generally intended to simplify the writing of programs to fall within the classes. We mention here the intrinsic characterizations of L and P by N. Jones [2, 3] which were our motivation. Our characterizations are designed for a possible separation and so they are stripped of all bells and whistles which make the programming easier. Although our definitions extensionally define the functions and predicates to be in the corresponding classes, their intensional use for computation is in general not very efficient mostly because we are not concerned with the degrees of polynomials hidden in the implicit time and space bounds and with the cost of a search through $2^{p(n)}$ values.

2 Grzegorczyk's Class of Functions $\mathcal{E}^2$

Our characterizations crucially depend on an intrinsic characterization of Grzegorczyk's class $\mathcal{E}^2$ by register machines whose programs have just two kinds of statements.

2.1 Function class $\mathcal{E}^2$. The class $\mathcal{E}^2$ of Grzegorczyk (see [1, 5]) consisting of functions over natural numbers ($\mathbb{N}$) can be generated from the *zero* function $Z(x) = 0$, *projection* functions:

$$I_i^n(x_1, \dots, x_i, \dots, x_n) = x_i \ ,$$

and from the *successor* function $S(x) = x + 1$ by *composition*:

$$f(\overline{x}) = g(h_1(\overline{x}), \dots, h_m(\overline{x})) \tag{1}$$

and by *polynomially limited primitive recursion*:

$$\begin{aligned} f(0, \overline{y}) &= g(\overline{y}) \\ f(x+1, \overline{y}) &= h(x, \overline{y}, f(x, \overline{y})) \\ f(x, \overline{y}) &\le p(x, \overline{y}) \end{aligned} \tag{2}$$

for a polynomial p. As a consequence, every function $f \in \mathcal{E}^2$ is *bounded* by a polynomial $p(\overline{x})$ such that

$$\forall \overline{x}\, f(\overline{x}) \le p(\overline{x}) \ . \tag{3}$$

2.2 Some simple polynomial bounds. For a polynomial $p(b)$ we have:

$$1 < b \to \exists d\, p(b) \le b^d \tag{1}$$

because if the degree of p is a then there is a b_0 such that $p(b) \le b^{a+1}$ for all $b > b_0$. For $1 < b \le b_0$ we have $p(b) \le p(b_0) \le b^c$ for some c. So it suffices to take $d = \max(a + 1, c)$.

To every $f \in \mathcal{E}^2$ and any numbers $c_1, \dots, c_n$ there is a number d such that

$$1 < b \wedge x_1 < b^{c_1} \wedge \dots \wedge x_n < b^{c_n} \to f(x_1, \dots, x_n) < b^d \ . \tag{2}$$

Indeed, there is a polynomial $p(x_1, \dots, x_n)$ such that for b and $x_1, \dots x_n$ as in the antecedent of (2) we have for some d:

$$f(x_1, \dots, x_n) \overset{2.1(3)}{\le} p(x_1, \dots, x_n) \le p(b^{c_1}, \dots, b^{c_n}) < p(b^{c_1}, \dots, b^{c_n}) + 1 \overset{(1)}{\le} b^d \ .$$

2.3 Some functions in $\mathcal{E}^2$. The basic arithmetic functions $x + y$, $x{\cdot}y$, $x \dot{-} y$ (*modified subtraction* where $x \dot{-} y = \max(0, x - y)$), $x \div y$, and $x \bmod y$ (*integer division and remainder* functions where for $y > 0$ we have $x = (x \div y){\cdot}y + x \bmod y$ and $x \div 0 = x \bmod 0 = 0$) are in $\mathcal{E}^2$ because the primitive recursion defining them is polynomially limited. For instance: $x{\cdot}0 = 0$, $x{\cdot}(y+1) = x{\cdot}y + x$ is limited by $x{\cdot}y \leq x{\cdot}y$. We can do case analysis in $\mathcal{E}^2$ with the ternary *if-then-else* function D satisfying $D(0, y, z) = z$ and $D(x + 1, y, z) = y$ defined by primitive recursion limited by $D(x, y, z) \leq y + z$. It is well-known that the class $\mathcal{E}^2$ is closed under bounded quantification and minimalization.

The binary exponentiation function b^k is not in $\mathcal{E}^2$ because it violates the polynomial bound. When k is a decimal constant then b^k is in $\mathcal{E}^2$ as a unary function in b with the definition:

$$b^k = D(b, 1 \overbrace{\cdot b \cdot \ldots \cdot b}^{k}, 0) \ .$$

Also the ternary function $\min(b^x, y)$ is in $\mathcal{E}^2$ by primitive recursion $\min(b^0, y) = \min(b^0, y)$ and $\min(b^{x+1}, y) = \min(b{\cdot}\min(b^x, y), y)$ limited by $\min(b^x, y) \leq y$.

2.4 Notation for numbers in base b. We will use the bounds from Par. 2.2 in the following by representing numbers in the base $b > 1$. For that we need some $\mathcal{E}^2$ functions working with such numbers.

We explicitly define a three argument function:

$$[x_1, x_2]^b = x_1{\cdot}b + x_2 \tag{1}$$

and extend the notation to $n \geq 4$ arguments by a series of definitions satisfying

$$[x_1, \ldots, x_n]^b = [[x_1, \ldots, x_{n-1}]^b, x_n]^b \ . \tag{2}$$

Provided that $x_1, \ldots, x_n < b$, the term $[x_1, \ldots, x_n]^b$ denotes the number whose n digits in the base b are $x_1, \ldots x_n$. For completeness sake we define $[x]^b = x$. All above functions are in $\mathcal{E}^2$.

We designate by $x[i]^b$ the ternary *base b indexing* function satisfying:

$$x[i]^b = (x \div b^i) \bmod b^i \ .$$

Despite the exponentiation, the function is in $\mathcal{E}^2$ because both functions $x \div b^i$ and $x \bmod b^i$ are in the class by $x \div b^i = x \div \min(b^i, x+1)$ and $x \bmod \min(b^i, x+1)$.

We designate by $x[i := v]^b$ the four place *modification* function satisfying

$$x[i := v]^b = (((x \div b^i) \div b){\cdot}b + v){\cdot}b^i + D(i, x \bmod b^i, 0) \ .$$

If $v < b$ then the function modifies the i-th digit of x by v. Because the second occurrence of the exponentiation b^i is unlimited, the function is not in $\mathcal{E}^2$. However, its variant $x[\min(i, c) := v]^b$ with c a decimal constant is in $\mathcal{E}^2$ because we can then replace the unlimited occurrence of b^i by $\min(b^i, b^c)$.

2.5 Dyadic notation. It is well-known that every positive natural number x can be uniquely represented in the *dyadic notation* as $x = \sum_{i<n} d_i{\cdot}2^i$ where the (dyadic) *digits* d_i are 1 or 2. We call the number n the *dyadic size of* x, in

symbols $|x|$. We will use the dyadic notation instead of binary for the coding of input and of Turing machine tapes because the dyadic notation has no problems with leading zeroes. As a consequence, the non-empty words over the two element alphabet $\{1,2\}$ can be uniquely coded by positive natural numbers whose dyadic digits code the symbols. The empty word is coded by the number 0. We have

$$2^{|x|} \leq x+1 < 2{\cdot}2^{|x|} \tag{1}$$

and so both the dyadic size function $|x|$ and $2^{|x|}$ are in $\mathcal{E}^2$ by:

$$\begin{aligned}|x| &= \mu_{s\leq x}[x+1 < 2{\cdot}\min(2^s, x+1)]\\ 2^{|x|} &= \min(2^{|x|}, x+1)\ .\end{aligned}$$

The *dyadic concatenation function* $x \star y$ is explicitly defined in $\mathcal{E}^2$ by:

$$x \star y = x{\cdot}2^{|y|} + y\ .$$

The *dyadic indexing* function $x[i]$ yielding the i-th dyadic digit of x if $i < |x|$ and 0 otherwise is in $\mathcal{E}^2$ by bounded minimalization:

$$x[i] = \mu_{d\leq 2}[1 \leq d \wedge \exists x_1{<}x \exists x_2{<}x(x = x_1 \star d \star x_2 \wedge |x_2| = i)]\ .$$

We will need the ternary *dyadic modification* function $x[i := d]$ which yields the number like x but with the i-th dyadic digit replaced by the digit d provided $i \leq |x|$ and $1 \leq d \leq 2$. The function yields 0 otherwise. The function is in $\mathcal{E}^2$ by bounded minimalization:

$$x[i := d] = \mu_{y\leq 2\star x}[1 \leq d \leq 2 \wedge \forall j{\leq}|x|(j = i \wedge y[j] = d \vee j \neq i \wedge y[j] = x[j])]\ .$$

Notice that when $i = |x|$ and $1 \leq d \leq 2$ we have $x[i := d] = d \star x$ and so the number x is extended with the dyadic digit d on the left.

3 Our Register and Turing Machine Models

3.1 Registers. Our programs operate on a finite number out of infinitely many *registers* designated by R_0, R_1, ... The registers contain natural numbers $i < b$ bounded by a number $b > 1$. We use the abbreviation $R_i^k \equiv R_{i+k-1}, \ldots, R_i$ to designate $k > 0$ consecutive registers ending in R_i. We can view the registers R_i^k as holding the digits of a number $< b^k$ represented in the base b (R_i holds the least significant digit).

We can naturally code the values of all registers accessible in a program by a single number x. The value of R_i is accessed as $x[i]^b$ and the same register is modified to a new value $v < b$ with $x[i := v]^b$.

3.2 Syntax of programs. *Programs* $\boldsymbol{\pi}$ are sequences of *statements* σ generated by the grammar:

$$\begin{aligned}\sigma &::= R_i := (R_i{+}1) \bmod b \mid \textbf{while}\, R_i{>}0\, \{\boldsymbol{\pi}\} \mid \\ &\quad\ \textbf{if}\ \boldsymbol{h}[R_i^h]\, \{\pi|\pi\} \mid \textbf{rand}\, \{\pi|\pi\} \\ \boldsymbol{\pi} &::= \sigma \mid \sigma;\ \boldsymbol{\pi}\end{aligned}$$

where $i, h \in \mathbb{N}$. The first type of statements are *incrementing assignments* increasing registers by 1 modulo b. The second type is a *while loop* with $\boldsymbol{\pi}$ its *body*. The third type are *history* statements which will be discussed in Par. 3.3. The last type of statements are *random* statements $\textbf{rand}\, \{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ nondeterministically executing either $\boldsymbol{\pi}_1$ or $\boldsymbol{\pi}_2$.

For a concrete program $\boldsymbol{\pi}$ we designate by the constant $L^{\boldsymbol{\pi}}$ the number $l+2$ where l is the number of statements in $\boldsymbol{\pi}$ and by $M^{\boldsymbol{\pi}}$ the number $m+1$ where R_m is the highest numbered register incremented in $\boldsymbol{\pi}$.

3.3 History statements. The idea of a history statement $\textbf{if}\ \boldsymbol{h}[R_i^h]\, \{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ appearing in a program $\boldsymbol{\pi}$ is to look to the closest history of the computation of $\boldsymbol{\pi}$ whether the *history* registers R_i^h contained the same values as now and to execute the programs $\boldsymbol{\pi}_1$ or $\boldsymbol{\pi}_2$ accordingly. The number h, which must be positive, is called the *dimension of history* and it is designated by the constant $D^{\boldsymbol{\pi}}$ (if $\boldsymbol{\pi}$ is without history statements then $D^{\boldsymbol{\pi}} = 0$).

If a program contains more than one history statements all must have the same numbers i and h. A history statement refers, in addition to the explicitly mentioned history registers R_i^h, also to two implicit registers: R_{i+h} called the *signal history* register and to R_{i+h+1} called the *reset history* register.

In order to avoid spurious results while the values of history registers are not yet fully assigned to we use the statement $R_{i+h} := (R_{i+h}{+}1) \bmod b$ when $R_{i+h} = 0$ to signal that the current values of history registers R_i^h should be remembered for a future reference by a history statement. The value 0 of the reset history register R_{i+h+1} at this time means that the values of history registers should be remembered as to be *set* whereas $R_{i+h+1} > 0$ means that the values should be remembered as to be *reset*.

We have included an example of use of the signal register to signal the setting of values in Par. 6.1 and to signal both the setting and resetting in Par. 7.1.

3.4 Syntactic classification of programs. Apart from the syntactic restriction that $D^{\boldsymbol{\pi}} > 0$ if the program $\boldsymbol{\pi}$ contains a history statement and that all history statements in $\boldsymbol{\pi}$ must refer to the same history registers, there are no additional restrictions on programs. Specifically, there are no semantic conditions imposed on programs by their operational semantics (see Par. 3.6).

We syntactically classify the programs as follows. Programs with increments and while loops only are *basic* programs. Programs without random statements are *deterministic*, or *d-programs*.

Programs π such that if $D^\pi > 0$ then they never increment their signal history registers are *simple* programs. Programs such that if $D^\pi > 0$ then they do not increment their reset history registers are *history setting* programs.

The table in Fig. 1 summarizes the claimed computation power of our programs. Proofs of the claims are given in Sections 4 to 7.

programs π	restrictions	bound on runtime	compute	proved in
basic	only incr. and whiles	$L^\pi \cdot b^{M^\pi}$	$\mathcal{E}^2$ †	Thm. 4.2
det. simple	no hist. signal, no rand.	$L^\pi \cdot b^{M^\pi}$	L	Thm. 5.3
simple	no hist. signal	$L^\pi \cdot b^{M^\pi}$	NL	Thm. 5.3
det. hist. setting	no hist. reset, no rand.	$L^\pi \cdot b^{D^\pi + M^\pi}$	P	Thm. 6.2
hist. setting	no hist. reset	$L^\pi \cdot b^{D^\pi + M^\pi}$	NP	Thm. 6.2
det.	no rand.	$L^\pi \cdot 2^{b^{D^\pi}} \cdot b^{M^\pi}$	*Pspace*	Thm. 7.2
any	none	$L^\pi \cdot 2^{b^{D^\pi}} \cdot b^{M^\pi}$	*Pspace*	Thm. 7.2

† with exponentially larger base b than the remaining characterizations.

Fig. 1. Computation power of various kinds of programs.

3.5 Assigning labels to programs. The standard development would now assign the meaning to a program π by a function $[\![\pi]\!]_b(x)$ taking the input values of registers x and yielding the output values of registers. Thus, for instance, we could define $[\![R_i := (R_i{+}1) \bmod b]\!]_b(x) = x[i := (x[i]^b{+}1) \bmod b]^b$. However, because of the history statements we need to assign the meaning to programs via arithmetization of their computation sequences.

Towards that end we assign to every statement σ of a program π a unique number l s.t. $0 \leq i < L^\pi$ as its *label*. The statements are labelled from left to right and $L^\pi \dot{-} 1$ labels the dummy statement at the end of π. For every program π there is a finite ternary relation $Follows^\pi(l, l_1, l_2)$ holding iff the labels l_1 and l_2 may follow the label l in a computation sequence for π. Thus if π has the form $\ldots R_i := (R_i{+}1) \bmod b;\ \pi_1 \ldots$ with the incrementing assignment labelled by l then $l_1 = l_2 = l + 1$.

If π has the form $\ldots$ **while** $R_i{>}0\,\{\pi_1\};\ \pi_2 \ldots$ and l is the label of the while loop then the label $l_1 = l + 1$ of π_1 is the destination of the positive outcome of the test and the label $l_2 = l + L^{\pi_1}$ of π_2 is the destination of the negative outcome of the test. We have also $Follows^\pi(l + L^{\pi_1} \dot{-} 1, l, l)$ because the loop statement is the destination after the last statement of the body of the loop.

If π has the form $\ldots$ **if** $\boldsymbol{h}[R_i^h]\,\{\pi_1|\pi_2\};\ \pi_3 \ldots$ and l is the label of the history statement then the label $l_1 = l{+}1$ of π_1 is the destination of the positive outcome of the test and the label $l_2 = l + L^{\pi_1}$ of π_2 is the destination of the negative outcome of the test. The label of the first statement of π_3 follows after the last statements of π_1 and π_2. The relation $Follows^\pi(l, l_1, l_2)$ is similar when l labels a random statement $\mathbf{rand}\,\{\pi_1|\pi_2\}$.

For every program π we also have finite predicates $Incr^\pi(l,i)$, $Whr^\pi(l,i)$, $Hist^\pi(l,i)$, and $Rand^\pi(l)$ holding iff l labels in that order $R_i := (R_i{+}1) \bmod b$, $\mathbf{while}\, R_i{>}0\,\{\pi_1\}$, $\mathbf{if}\, \boldsymbol{h}[R_i^h]\,\{\pi_1|\pi_2\}$, and $\mathbf{rand}\,\{\pi_1|\pi_2\}$.

$$
\begin{aligned}
&Last^\pi(q) = \mu_{n\le q}[q \div (L^\pi)^{n+1} = 0]\\
&R_b^\pi(q,0,x) = x\\
&R_b^\pi(q,j{+}1,x) = \begin{cases} y[i := (y[i]^b{+}1) \bmod b]^b & Incr^\pi(q[j]^{L^\pi}, i) \land R_b^\pi(q,j,x) = y\\ R_b^\pi(q,j,x) & \text{otherwise}\end{cases}\\
&Next_b^\pi(q,j,x,l) \leftrightarrow \exists l_1 \exists l_2 \exists i (l_1 = q[j]^{L^\pi} \land (\\
&\quad Follows^\pi(l_1,l,l_2) \land \exists y(y = R_b^\pi(q,j{+}1,x) \land (\\
&\qquad Incr^\pi(l_1,i) \lor\\
&\qquad Whr^\pi(l_1,i) \land y[i]^b > 0 \lor\\
&\qquad Hist^\pi(l_1,i) \land Hs_b^\pi(q,j,x,y,i) \lor\\
&\qquad Rand^\pi(l_1)\,))\lor\\
&\quad Follows^\pi(l_1,l_2,l) \land \exists y(y = R_b^\pi(q,j{+}1,x) \land (\\
&\qquad Whr^\pi(l_1,i) \land y[i]^b = 0 \lor\\
&\qquad Hist^\pi(l_1,i) \land \neg Hs_b^\pi(q,j,x,y,i) \lor\\
&\qquad Rand^\pi(l_1)\,))\ ))\\
&H_b^\pi(q,j,x,y,i) = \mu_{m\le j}[(R_b^\pi(q,j\dot{-}m,x))[i{+}D^\pi]^b = 1 \land\\
&\qquad\qquad R_b^\pi(q,j\dot{-}m,x) \div b^i \bmod b^{D^\pi} = y \div b^i \bmod b^{D^\pi}]\\
&Hs_b^\pi(q,j,x,y,i) \leftrightarrow \exists m(m = H_b^\pi(q,j,x,y,i) \land m > 0 \land (R_b^\pi(q,j\dot{-}m,x))[i{+}D^\pi{+}1]^b = 0)\\
&Cseq_b^\pi(q,x,y) \leftrightarrow \exists n(n = Last^\pi(q) \land q[0]^{L^\pi} = 0 \land q[n]^{L^\pi} = L^\pi \dot{-} 1 \land\\
&\qquad\qquad \forall j{<}n\, Next_b^\pi(q,j,x,q[j{+}1]^{L^\pi}) \land R_b^\pi(q,n,x) = y)
\end{aligned}
$$

Fig. 2. Computation sequences for programs.

3.6 Operational semantics of programs. For a given program π its computation sequence q satisfies the four place predicate $Cseq^\pi$ given in Fig. 2. Notice that to every π there is a predicate $Cseq^\pi$ where π is not a parameter but rather a part of the predicate symbol.

The *computation sequence* q s.t. $Cseq_b^\pi(q,x,y)$ is a number whose j-th element $q[j]^{L^\pi}$ holds the label of a statement of π executed at the j-th step of its execution. The execution starts with the initial label 0 in the step 0 and with the initial values of registers x. The execution ends with the final label $L-1$ in the step $Last^\pi(q)$ with final values of registers y. When π started with x does not

instruction	meaning
$\mathit{cleft}_x(q_1, q_2)$	$\boldsymbol{if}\ i+1<\lvert x\rvert\ \boldsymbol{then}\ i := i+1;\ \boldsymbol{goto}\, q_1\ \boldsymbol{else}\ \boldsymbol{goto}\, q_2$
$\mathit{left}_t(q)$	$w := w+1;\ \boldsymbol{goto}\, q$
$\mathit{cright}_x(q_1, q_2)$	$\boldsymbol{if}\ i>0\ \boldsymbol{then}\ i := i \dot{-} 1;\ \boldsymbol{goto}\, q_1\ \boldsymbol{else}\ \boldsymbol{goto}\, q_2$
$\mathit{cright}_t(q_1, q_2)$	$\boldsymbol{if}\ w>0\ \boldsymbol{then}\ w := w \dot{-} 1;\ \boldsymbol{goto}\, q_1\ \boldsymbol{else}\ \boldsymbol{goto}\, q_2$
$\mathit{if}_x(q_1, q_2, q_3)$	$\boldsymbol{if}\ x[i]=2\ \boldsymbol{then}\ \boldsymbol{goto}\, q_1\ \boldsymbol{else}\ \boldsymbol{if}\ x[i]=1\ \boldsymbol{then}\ \boldsymbol{goto}\, q_2\ \boldsymbol{else}\ \boldsymbol{goto}\, q_3$
$\mathit{if}_t(q_1, q_2, q_3)$	$\boldsymbol{if}\ t[w]=2\ \boldsymbol{then}\ \boldsymbol{goto}\, q_1\ \boldsymbol{else}\ \boldsymbol{if}\ t[w]=1\ \boldsymbol{then}\ \boldsymbol{goto}\, q_2\ \boldsymbol{else}\ \boldsymbol{goto}\, q_3$
$\mathit{write}_t(s, q)$	$t := t[w := s];\ \boldsymbol{goto}\, q$ where $s = 1, 2$
$\mathit{rand}(q_1, q_2)$	go nondeterministically to label q_1 or q_2

Fig. 3. Instructions of Turing machines.

terminate there is no computation sequence because infinite sequences cannot be coded in this way.

The four place function $R^{\boldsymbol{\pi}}$ applied as $R^{\boldsymbol{\pi}}_b(q, j, x)$ reads off the computation sequence q the labels of the increment statements (the only ones which change registers), computes their effect, and yields the value of registers just before the j-step of execution of $\boldsymbol{\pi}$ which has started with the initial values of registers x.

The five place predicate $\mathit{Next}^{\boldsymbol{\pi}}_b(q, j, x, l)$ holds for q s.t. $\mathit{Cseq}^{\boldsymbol{\pi}}_b(q, x, y)$ and $j < \mathit{Last}^{\boldsymbol{\pi}}(q)$ iff l is the label of $\boldsymbol{\pi}$ to be executed in the step $j+1$. $\mathit{Next}^{\boldsymbol{\pi}}$ calls on the predicate $\mathit{Hs}^{\boldsymbol{\pi}}$ implementing the history statements as explained in Par. 3.3.

3.7 Turing machines accepting unary predicates over $\mathbb{N}$. Our Turing machines compute unary predicates $P(x)$ over $\mathbb{N}$. We define the Turing machines slightly non-standardly (but without altering their time and space bounds for the complexity classes we are interested in) chiefly to avoid the complications of simulating them when they walk off the tape.

The *input tape* is in the alphabet $\{0, 1, 2, \triangleleft\}$ where the arguments x are presented in the dyadic notation as words $d_{|x|} \cdots d_0 \triangleleft$ where $d_{|x|} = 0$ and we have $x = d_{|x|} \star \cdots \star d_0$. We use the end markers 0 and $\triangleleft$ to ensure that the input scanning position i always satisfies $0 \le i \le |x| \dot{-} 1$ with the scanned symbol d_i.

The input is processed on a *work tape* t with the same tape symbols and form as the input tape. The work tape scanning position w always stays as $0 \le w \le |t|$. We can assume w.l.o.g. that the scanning position is never increased beyond $|t|$ which means that the leftmost 0 must be first overwritten by 1 or 2 before the machine moves left.

A *program* for a Turing machine M is a finite number of pairs $q : i$ where $q > 0$ is a label (state) and i is an instruction. All instructions must have different labels. An attempt to go to a label q which is not in a program halts the machine with a rejection unless $q = 0$ when the machine accepts. The configuration of M is given by the quintuple $\langle q, x, i, t, w\rangle$ where q is the label of an instruction to be executed, x is the code of the input tape with i its scanning position, and t the code of the work tape with w its scanning position. Figure 3 describes the instructions and their effect.

Nondeterministic Turing machines have additional instructions $rand(q_1, q_2)$ transferring the control randomly to labels q_1 or q_2.

A Turing machine M (deterministic or not) computing the predicate $P(x)$ starts with the configuration $\langle 1, x, 0, 0, 0\rangle$. We assume that the machine stops on every input by accepting or rejecting. $P(x)$ is true iff at least one execution path ends with the success label 0.

The time and space bounds on our Turing machines are defined as usual.

4 Characterization of Basic Programs

The main result Thm. 4.2 of this section characterizes the basic programs as computing exactly the functions and predicates of $\mathcal{E}^2$.

4.1 Macros. A *macro* is a named program with parameters like register numbers i or programs $\boldsymbol{\pi}$ which can be used with concrete values. All macros in this paragraph are basic programs.

The following defines a macro $R_i := 0$ with one parameter i clearing the register R_i:

$$R_i := 0 \equiv \textbf{while}\, R_i{>}0\, \{R_i := (R_i{+}1) \bmod b\}\ .$$

The macro $R_i := 1$ sets 1 into the register R_i:

$$R_i := 1 \equiv R_i := 0;\ R_i := (R_i{+}1) \bmod b\ .$$

Notice that the macro works properly only if $b > 1$ which will be always the case when executing programs. The macro uses the register clearing macro.

In the following we will slightly abuse the notation by writing $\mu;\ \boldsymbol{\pi}$ for the instantiation of a macro μ when μ has more than one statement. In such a case we should concatenate $\boldsymbol{\pi}$ after μ by $\mu \oplus \boldsymbol{\pi}$ where the concatenation of programs $\oplus$ is defined to satisfy: $\sigma \oplus \boldsymbol{\pi}_1 \equiv \sigma;\ \boldsymbol{\pi}_1$ and $(\sigma;\ \boldsymbol{\pi}) \oplus \boldsymbol{\pi}_1 \equiv \sigma;\ (\boldsymbol{\pi} \oplus \boldsymbol{\pi}_1)$.

We can assign from register R_j to register R_i by the macro $R_i := R_j$:

$$\begin{aligned}R_i := R_j \equiv R_a := 0;\ &\textbf{while}\, R_j{>}0\, \{R_a := (R_a{+}1) \bmod b;\ R_j := (R_j{+}1) \bmod b\};\\ R_i := 0;\ &\textbf{while}\, R_a > 0\, \{R_i := (R_i{+}1) \bmod b;\ R_j := (R_j{+}1) \bmod b;\\ &\qquad R_a := (R_a{+}1) \bmod b\}\ .\end{aligned}$$

That the assignment macro works properly can be seen from the fact that if $R_j = n$ before the first while loop we have $R_a \equiv b - n \pmod b$ and $R_j = 0$ after it. Although the assignment macro keeps the value of R_j unchanged, it destroys an auxiliary register R_a. In the following we will tacitly assume that the auxiliary registers are instantiated without conflicts when macros are used. A conditional can be defined with two auxiliary registers as follows:

$$\begin{aligned}\textbf{if}\, R_i > 0\{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\} \equiv R_a := R_i;\ R_c := 1;\ &\textbf{while}\, R_a{>}0\, \{\boldsymbol{\pi}_1;\ R_a := 0;\ R_c := 0\};\\ &\textbf{while}\, R_c{>}0\, \{\boldsymbol{\pi}_2;\ R_c := 0\}\ .\end{aligned}$$

Increment with carry macro is programmed as follows:

$$R_c, R_i := R_i{+}1 \equiv R_i := (R_i{+}1) \bmod b;\ \mathbf{if}\, R_i > 0\,\{R_c := 0 \mid R_c := 1\}\ .$$

The macro is generalized to arbitrary number of registers R_i^d with $d \geq 1$ by a sequence of macros:

$$\begin{aligned} R_c, R_i^{d+1} := R_i^{d+1}{+}1 \equiv R_c, R_i^d := R_i^d{+}1; \\ \mathbf{if}\, R_c > 0\,\{R_c, R_{i+d} := R_{i+d}{+}1 \mid R_c := 0\ .\} \end{aligned}$$

We leave to the reader the programming of a macro: $R_i := R_{j_1}^d <_* R_{j_2}^d$ which computes into R_i the characteristic function (1 true, 0 false) of the relation $R_{j_1}^d < R_{j_2}^d$ and also of multiple assignment macros $R_i^d := 0$ and $R_i^d := R_j^c$ with $d \geq c \geq 1$. We will need the following multiple assignment macro where $d \geq c_1 + c_2$ and $c_1, c_2 \geq 1$:

$$R_i^d := R_{j_1}^{c_1}, R_{j_2}^{c_2} \equiv R_i^{c_2} := R_{j_2}^{c_2};\ R_{i+c_2}^{d-c_2} := R_{j_1}^{c_1}\ .$$

We can enter an infinite loop with the macro:

$$\mathit{Fail} \equiv R_0 := 1;\ \mathbf{while}\, R_0{>}0\,\{R_0 := 1\}\ .$$

Finally, will use the following macros for the setting and resetting of history:

$$\begin{aligned} \mathbf{set}\ \boldsymbol{h}[R_i^h] &\equiv R_{i+h} := (R_{i+h}{+}1) \bmod b;\ R_{i+h} := (R_{i+h}{+}1) \bmod b;\ R_{i+h} := 0 \\ \mathbf{reset}\ \boldsymbol{h}[R_i^h] &\equiv R_{i+h+1} := (R_{i+h+1}{+}1) \bmod b;\ \mathbf{set}\ \boldsymbol{h}[R_i^h];\ R_{i+h+1} := 0\ . \end{aligned}$$

4.2 Lemma ($\mathcal{E}^2$ functions have macros). *To every n-ary $f \in \mathcal{E}^2$ and any numbers $c_1, \ldots, c_n, d \geq 1$ such that 2.2(2) holds there is a basic macro $\boldsymbol{\pi} \equiv R_i^d := f(R_{j_1}^{c_1}, \ldots, R_{j_n}^{c_n})$ which with the arguments $\overline{x}$ placed into the corresponding registers $R_{j_k}^{c_k}$ on input leaves the value of f in the registers R_i^d on output. This is formally expressed by:*

$$\begin{aligned} 1 < b \wedge x_1 = v \div b^{j_1} \bmod b^{c_1} \wedge \cdots \wedge x_n = v \div b^{j_n} \bmod b^{c_n} \rightarrow \\ (\, f(x_1, \ldots, x_n) = y \leftrightarrow \exists q \exists w (\mathit{Cseq}_b^{\boldsymbol{\pi}}(q, v, w) \wedge y = w \div b^i \bmod b^d)\,)\ . \end{aligned}$$

Proof. By induction on the construction of f. If $f = Z$ then define:

$$R_i^d := Z(R_j^{c_1}) \equiv R_i^d := 0\ .$$

If $f = I_i^n$ then define:

$$R_i^d := I_i^n(R_{j_1}^{c_1}, \ldots, R_{j_n}^{c_n}) \equiv R_i^d := R_{j_i}^d\ .$$

If $f = S$ then it must be the case that $c_1 < d$ and so we define:

$$R_i^d := S(R_{j_1}^{c_1}) \equiv R_i^d := R_{j_1}^{c_1};\ R_{i+c_1}, R_i^{c_1} := R_i^{c_1}{+}1\ .$$

If f is defined by composition 2.1(1) then we obtain from 2.2(2) bounds $d_1, \ldots, d_m$ for the functions $h_1, \ldots h_m$ respectively and from m IH's macros for the functions. For g and the numbers $d_1, \ldots, d_m$ we obtain from 2.2(2) a bound d_0 for g. We then use the bound $\max(d_0, d)$ in another IH to obtain a macro for g so we can define:

$$R_i^d := f(R_{j_1}^{c_1}, \ldots, R_{j_n}^{c_n}) \equiv R_{p_1}^{d_1} := h_1(R_{j_1}^{c_1}, \ldots, R_{j_n}^{c_n});\ \ldots;\ R_{p_m}^{d_m} := h_m(R_{j_1}^{c_1}, \ldots, R_{j_n}^{c_n});$$
$$R_q^{\max(d_0,d)} := g(R_{p_1}^{d_1}, \ldots, R_{p_m}^{d_m});\ R_i^d := R_q^d$$

with auxiliary registers $R_{p_1}^{d_1}, \ldots, R_{p_m}^{d_m}$, and $R_p^{\max(d_0,d)}$.

If $f(x, \overline{y})$ is defined by limited primitive recursion 2.1(2) we obtain for $c_2, \ldots, c_{n+1}$ a bound d_1 on $g(\overline{y})$ from 2.2(2) and use the bound $\max(d_1, d)$ in IH to get a macro for g. For $c_1, c_2, \ldots, c_{n+1}$, and d we obtain a bound d_2 on $h(x, \overline{y}, r)$ from 2.2(2) and use the bound $d_0 = \max(d_2, d_1, d)$ in IH to obtain a macro for h. For the auxiliary registers R_a, $R_p^{c_1}$ (counting from 0 to $R_{j_1}^{c_1}$), and $R_q^{d_0}$ (holding $f(R_p^{c_1}, R_{j_2}^{c_2}, \ldots, R_{j_{n+1}}^{c_{n+1}})$) we then define:

$$\begin{aligned}
R_i^d &:= f(R_{j_1}^{c_1}, R_{j_2}^{c_2}, \ldots, R_{j_{n+1}}^{c_{n+1}}) \equiv \\
&\quad R_q^{d_0} := 0;\ R_q^{\max(d_1,d)} := g(R_{j_2}^{c_2}, \ldots, R_{j_{n+1}}^{c_{n+1}});\ R_p^{c_1} := 0;\ R_a := R_p^{c_1} <_* R_{j_1}^{c_1}; \\
&\quad \textbf{while}\, R_a > 0\, \{R_q^{d_0} := h(R_p^{c_1}, R_{j_2}^{c_2}, \ldots, R_{j_{n+1}}^{c_{n+1}}, R_q^d); \\
&\qquad\qquad R_a, R_p^{c_1} := R_p^{c_1} + 1;\ R_a := R_p^{c_1} <_* R_{j_1}^{c_1}\}; \\
&\quad R_i^d := R_q^d\ . \qquad \square
\end{aligned}$$

4.3 Functions and predicates computed by basic programs. We say that the n-ary function f is *computed* by the basic program π if we have

$$f(\overline{x}) = y \leftrightarrow \exists b \exists q (b = \max(1, \overline{x}) + 1 \land Cseq_b^{\pi}(q, [\overline{x}]^b, y))\ . \tag{1}$$

Notice that the input $\overline{x}$ to the basic program is put into the corresponding registers R_0^n and the output is left in the registers R_0^d for some constant d with the remaining registers cleared.

The n-ary predicate P is *computed* by the basic program π if we have

$$P(\overline{x}) \leftrightarrow \exists b \exists q \exists y (b = \max(1, \overline{x}) + 1 \land Cseq_b^{\pi}(q, [\overline{x}]^b, y))\ . \tag{2}$$

Notice that the program π computing P terminates for the input $\overline{x}$ in the corresponding registers R_0^n iff $P(\overline{x})$ holds.

4.4 Theorem (Characterization of basic programs). *Basic programs compute exactly the functions and predicates of $\mathcal{E}^2$.*

Proof. Let π be a basic program. We cannot directly use the definition 4.3(1) to define in $\mathcal{E}^2$ the n-ary function f computed by π because we have only an exponential bound $(L^\pi)^{L^\pi \cdot b^{M^\pi}}$ on the sequence q. We can, however, obtain the j-th element of the sequence by a ternary function $g_b^j(x)$ satisfying:

$$\max(1,\overline{x}) < b \wedge \mathit{Cseq}_b^\pi(q, [\overline{x}]^b, y) \wedge j \leq \mathit{Last}^\pi(q) \rightarrow \\ g_b^j([\overline{x}]^b) = q[j]^{L^\pi} \cdot b^c + R_b^\pi(q, j, [\overline{x}]^b)$$

where we abbreviate by c the constant $\max(n, M^\pi)$. The function g is in

$$\begin{aligned}
&\mathit{Nxt}_b(l_1, y, l, z) \leftrightarrow \exists l_2 < L^\pi \exists i < M^\pi(\\
&\quad l_1 = L^\pi \dot{-} 1 \wedge z = y \wedge l = l_1 \vee \\
&\quad \mathit{Follows}^\pi(l_1, l, l_2) \wedge (\mathit{Incr}^\pi(l_1, i) \wedge z = y[\min(i,c) := (y[i]^b {+} 1) \bmod b]^b \vee \\
&\qquad\qquad \mathit{Whr}^\pi(l_1, i) \wedge y[i]^b > 0 \wedge z = y) \vee \\
&\quad \mathit{Follows}^\pi(l_1, l_2, l) \wedge \mathit{Whr}^\pi(l_1, i) \wedge y[i]^b = 0 \wedge z = y)
\end{aligned}$$

$$g_b^0(x) = x \bmod b^c$$

$$g_b^{j+1}(x) = \begin{cases} l \cdot b^c + z & g_b^j(x) = l_1 \cdot b^c + y \wedge y < b^c \wedge \\ & z < b^c \wedge l < L^\pi \wedge \mathit{Nxt}_b(l_1, y, l, z) \\ 0 & \text{otherwise} \end{cases}$$

$$g_b^j(x) < L^\pi \cdot b^c + b^c$$

Fig. 4.

$\mathcal{E}^2$ (see Fig. 4) by polynomially limited primitive recursion from a predicate $\mathit{Nxt}_b(l_1, y, l, z)$ holding iff π executing l_1 with registers y goes to label l with registers z. The predicate is obtained by a slight modification for basic programs of R^π and Next^π from Fig. 2. The modification was also needed to get the predicate into $\mathcal{E}^2$.

We can now define $f \in \mathcal{E}^2$ by explicit definition:

$$f(\overline{x}) = g_{\max(1,\overline{x})+1}^{L^\pi \cdot b^{M^\pi}}([\overline{x}]^{\max(1,\overline{x})+1}) \bmod b^c \ .$$

Notice that if the final label $L^\pi \dot{-} 1$ is not reached in $L^\pi \cdot b^{M^\pi}$ iterations of g, it never will.

Vice versa, for an n-ary $f \in \mathcal{E}^2$ and the numbers $c_1 = \cdots = c_n = 1$ we obtain a bound d from 2.2(2) and use it in Lemma 4.2 to get a macro: $R_i^d := f(R_{j_1}, \ldots, R_{j_n})$. We construct the basic program π computing f as follows:

$$\pi \equiv R_0^d := f(R_0, \ldots, R_n);\ R_d^{M^\pi} := 0 \ .$$

If the basic program π computes an n-ary predicate P then we construct the iteration function g as above and define the predicate P to be in $\mathcal{E}^2$ by

$$P(\overline{x}) \leftrightarrow \exists y {<} b^c\, g_{\max(1,\overline{x})+1}^{L^\pi \cdot b^{M^\pi}}([\overline{x}]^{\max(1,\overline{x})+1}) = (L^\pi \dot{-} 1) \cdot b^c + y \ .$$

Vice versa, given an n-ary predicate P in $\mathcal{E}^2$ we obtain a number d and a macro $R_i^d := f(R_{j_1}, \dots, R_{j_n})$ computing its characteristic function f as above. We construct the basic program $\boldsymbol{\pi}$ computing P by converting the false value of f into a non-terminating loop:

$$\boldsymbol{\pi} \equiv R_0^d := f(R_0, \dots, R_n);\ \mathbf{if}\, R_0 > 0 \{ R_0 := (R_0{+}1) \bmod b \mid \mathit{Fail} \}\ . \qquad \square$$

4.5 Remark. R. Ritchie [4] proved that the predicates of $\mathcal{E}^2$ (i.e. their characteristic functions) are exactly the ones computed in *Linspace*. J. Warkentin [6, 7] investigated two classes of functions bounded by $\max(k, \overline{x})$ for some constant k which are similar to the functions computed by our basic programs. He proved that the predicates of the two classes coincide with the predicates of $\mathcal{E}^2$. The characterization of functions computable by basic programs seems to be ours.

5 Characterization of Simple Programs

The characterization of simple programs in this section deals with the sublinear complexity classes L and NL and so our programs will not be able to receive their input in registers because their capacity will be limited by the size $|x| = b$ of the input x. The straightforward extension of basic programs to process input would be by input statements $\mathbf{if}\, x[R_i^k]{=}2\, \{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}$ testing the dyadic digit $x[[R_i^k]^b]$ of input. We will explore the input statements in Sect. 8. However, in order to allow for the characterization of classes with polynomial memory, we will until then access input through memory simulated by allowing the programs to peek into their history.

5.1 Input as prehistory. We treat the input x to programs which is coded in the dyadic notation as if it were a $k \geq 1$ dimensional Pascal-like array:

$$x : \boldsymbol{array}[0 \mathinner{..} b-1, \dots, 0 \mathinner{..} b-1]\ \boldsymbol{of}\ [1 \mathinner{..} 2]\ .$$

The digits of x are accessed by $x[[y_1, \dots, y_k]^b]$. The standard coding of input is one-dimensional but the classes we are interested in are robust enough to allow input of any dimension.

For a program $\boldsymbol{\pi}$ with history statements $\mathbf{if}\ \boldsymbol{h}[R_i^h]\, \{\boldsymbol{\pi}_1 | \boldsymbol{\pi}_2\}$ we expect $0 < k \leq h = D^{\boldsymbol{\pi}}$ to hold. Since, the number k cannot be read off the program $\boldsymbol{\pi}$, we will use the pair $\langle \boldsymbol{\pi}, k \rangle$, abbreviated as $\boldsymbol{\pi}^k$, to designate the program $\boldsymbol{\pi}$ intended to have the input of dimensionality k.

We treat the input x to $\boldsymbol{\pi}^k$ as if it were remembered in its *prehistory*. This means that whenever a history statement $\mathbf{if}\ \boldsymbol{h}[R_i^h]\, \{\boldsymbol{\pi}_1 | \boldsymbol{\pi}_2\}$ of $\boldsymbol{\pi}$ is executed with $R_{i+k}^{h-k} = 0$ then the statement refers to the digits of x and its test succeeds iff the dyadic digit $[R_i^h]^b = [R_i^k]^b$ of x is 2, i.e. iff $x[[R_i^k]^b] = 2$.

The input x to $\boldsymbol{\pi}$ is set into the prehistory by a program yielding function $\beta^{\boldsymbol{\pi}}(x)$. The program $\boldsymbol{\pi}$ is then executed with all its registers cleared and with its prehistory attached as $\beta^{\boldsymbol{\pi}}(x); \boldsymbol{\pi}$. The function β (see Fig. 5) calls on a ternary

$$\alpha_i^h(0) \equiv R_i^h := 0$$
$$\alpha_i^h(2{\cdot}x+1) \equiv \alpha_i^h(x);\ R_{i+h+2}, R_i^h := R_i^h + 1$$
$$\alpha_i^h(2{\cdot}x+2) \equiv \alpha_i^h(x);\ \textbf{set } \boldsymbol{h}[R_i^h];\ R_{i+h+2}, R_i^h := R_i^h + 1$$
$$\beta^{\pi}(x) \equiv \begin{cases} \alpha_i^h(x);\ R_{i+h+2} := 0;\ R_i^h := 0 & \text{if } h = D^{\pi} > 0 \text{ and } \boldsymbol{\pi} \text{ has hist. regs. } R_i^h \\ R_0 := 0 & \text{otherwise.} \end{cases}$$

Fig. 5. Metamathematical functions constructing the prehistory.

program yielding function $\alpha_i^h(x)$ which sets into history in the registers R_i^h all digits 2 of the dyadic representation of x.

If $\boldsymbol{\pi}$ is a simple program then, since no history can be set in $\boldsymbol{\pi}$, its history statements can only refer to the input via the prehistory. This restriction seems to be crucial because by Thm. 5.3 simple d-programs compute exactly the predicates in L. However, the example in Par. 6.1 demonstrates that a deterministic history setting program $\boldsymbol{\pi}^2$ can compute the transitive closure of a directed graph (which is probably not in L) by adding more edges into its input graph.

5.2 Predicates computed by programs. We intend to compute unary predicates $P(x)$ by programs $\boldsymbol{\pi}^k$ with k-dimensional input x. We will thus have $b^k = |x|$. Execution of $\boldsymbol{\pi}$ makes sense only when $b > 1$, i.e. $x > 2$, and so we must compute the values of $P(0)$, $P(1)$, and $P(2)$ differently. We call a unary predicate R *small* if $R(x) \to x \le 2$ holds. There are eight small predicates which we will use to define $P(x)$ for $x \le 2$. These considerations lead to the following definition.

Given a program $\boldsymbol{\pi}^k$ and a small predicate P_1 we say that the unary predicate $P(x)$ is *computed by* $\boldsymbol{\pi}^k$ *and* P_1 if we have:

$$P(x) \leftrightarrow \exists b \exists q \exists y (\, b^k = |x| \land b > 1 \land Cseq_b^{\beta^{\pi}(x);\, \boldsymbol{\pi}}(q,0,y)\,) \lor P_1(x)\ . \qquad (1)$$

5.3 Theorem (Characterization of simple programs). *Simple d-programs (programs) compute exactly the predicates in L (NL). Already the programs with* $D^{\pi} = 1$ *are such.*

Proof. For the proof that the predicates of L (NL) are computed by simple d-programs (programs) with $D^{\pi} = 1$ we take a Turing machine M accepting a predicate $P(x)$ in L (NL) with its work tape t and its scanning position w at all times such that $w, |t| \le k_1{\cdot}||x|| + k_2$ for some constants k_1, k_2. When $x \le 2$ we compute P from the small predicate $P_1(x) \leftrightarrow P(x) \land x \le 2$. When $x > 2$ we set $b = |x| > 1$ and so

$$t \overset{2.5(1)}{<} 2{\cdot}2^{|t|} \le 2{\cdot}2^{k_1\cdot|b|+k_2} = 2^{k_2+1}{\cdot}(2^{|b|})^{k_1} \overset{2.5(1)}{\le} 2^{k_2+1}{\cdot}(b+1)^{k_1} \overset{2.2(1)}{\le} b^{c_1}$$

for some c_1. We also have $w \le |t| \le k_1{\cdot}|b| + k_2 < b^{c_2}$ for some c_2.

We construct the simple d-program (program) $\boldsymbol{\pi}^1$ with $D^{\boldsymbol{\pi}} = 1$ simulating M with one history register R_0 where $\boldsymbol{\pi}$ holds the currently scanned position i of M's input tape x. Statements **if** $\boldsymbol{h}[R_0^1]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ execute $\boldsymbol{\pi}_1$ or $\boldsymbol{\pi}_2$ depending on whether $x[i] = 2$ or not. The work tape t of M is accessed by three $\mathcal{E}^2$ functions $t[w]$, $t[w := 1]$, and $t[w := 2]$ which for $t < b^{c_1}$ and $w < b^{c_2}$ are bounded by b^2, b^{c_1}, and b^{c_1} respectively. We use the bounds in Lemma 4.2 to obtain the macros

$$R_i^2 \mathrel{\boldsymbol{:=}} R_{j_1}^{c_1}[R_{j_2}^{c_2}] \quad R_i^{c_1} \mathrel{\boldsymbol{:=}} R_{j_1}^{c_1}[R_{j_2}^{c_2} := 1] \quad R_i^{c_1} \mathrel{\boldsymbol{:=}} R_{j_1}^{c_1}[R_{j_2}^{c_2} := 2]\ .$$

The program $\boldsymbol{\pi}$ can thus store the symbol currently scanned on the work tape in the registers R_3^2 (remember that it can be the case that $b \dot{-} 1 = 1$), the work tape t in the registers $R_5^{c_1}$, and the work tape scanning position in the registers $R_{5+c_1}^{c_2}$. It is now a straightforward programming to simulate the instructions of M in Fig. 3. If M accepts then $\boldsymbol{\pi}$ terminates and if M rejects then $\boldsymbol{\pi}$ goes into an infinite loop. Thus the predicate P is computed by $\boldsymbol{\pi}$ and P_1.

Vice versa, we take a predicate $P(x)$ computed by a simple d-program (program) $\boldsymbol{\pi}^k$ and by a small predicate P_1 and we construct a Turing machine M accepting $P(x)$ in L (NL) as follows. M determines first the length $|x|$ of the input by counting the number of left moves on its input tape until the end marker 0 is reached. If $|x| \leq 1$ then M accepts or rejects according to $P_1(x)$. Otherwise M determines whether $b^k = |x|$ for some b (this can be done in space $\mathcal{O}(||x||)$) and rejects if not. M uses its work tape to store the finite number of registers which are incremented in $\boldsymbol{\pi}$. Each register holds numbers less than b and it is initially cleared. M then simulates the program $\boldsymbol{\pi}$ whose statements, except perhaps **if** $\boldsymbol{h}[R_i^{D^{\boldsymbol{\pi}}}]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$, pose no problem. Simple programs cannot increment the signal history register $R_{i+D^{\boldsymbol{\pi}}}$. Thus whenever $D^{\boldsymbol{\pi}} > k$ and $R_{i+k}^{D^{\boldsymbol{\pi}}-k} > 0$ the test of history must fail and so it suffices for M to simulate $\boldsymbol{\pi}_2$. Otherwise the history statement refers to the prehistory which M reads off its input tape. To that end M computes the index $[R_i^k]^b < |x|$ into a separate counter j, sets the input tape to the rightmost position, and advances j times to the left. M then executes $\boldsymbol{\pi}_1$ or $\boldsymbol{\pi}_2$ depending on whether the currently scanned symbol is 2 or 1.

If $P(x)$ holds then the simulation of $\boldsymbol{\pi}$ terminates and M accepts. M must reject when $P(x)$ does not hold. History is never set by $\boldsymbol{\pi}$ and thus if $\boldsymbol{\pi}$ does not terminate in $L^{\boldsymbol{\pi}} \cdot b^{M^{\boldsymbol{\pi}}}$ steps it never will. M therefore maintains an execution timer and when the simulation times out M rejects.

M operates in L (NL) because its finite number of counters and registers need $\mathcal{O}(||x||)$ space. $\square$

5.4 Remark. The technique of Warkentin mentioned in Par. 4.5 was independently used in [2, 3] by N. Jones in his characterization of L. His counter machines are first bounded by polynomials and then reduced to the length of input (as in Lemma 4.2). Our basic programs are inspired by the counter machines of Jones except that we reduce their instructions to the bare minimum. This is possible because we are not interested in the most efficient programs.

Jones uses a special input register. Our contribution is that by shifting the input into prehistory we obtain a uniform characterization by the access to history of all classes in the 'enigmatic hierarchy' $L \subseteq NL \subseteq P \subseteq NP \subseteq Pspace$.

6 Characterization of Programs Setting History

The situation changes when we move from simple programs to programs incrementing their signal but not the reset history registers. We will prove in Thm. 6.2 that the history setting d-programs (programs) compute exactly P (NP).

$$\mathbf{forall}\, R_z\, \{\mathbf{forall}\, R_x\, \{\mathbf{if}\, P(R_x,R_z)\, \{\mathbf{forall}\, R_y\, \{\mathbf{if}\, P(R_z,R_y)\, \{\mathbf{set}\, P(R_x,R_y)\}\}\}\}\}$$

Used macros:

$$\begin{aligned}
&\mathbf{set}\, P(R_i,R_j) \equiv R_m^2 := R_i, R_j;\ \mathbf{set}\ \boldsymbol{h}[R_m^2] \\
&\mathbf{if}\, P(R_i,R_j)\,\{\boldsymbol{\pi}\} \equiv R_m^2 := R_i, R_j;\ \mathbf{if}\ \boldsymbol{h}[R_m^2]\,\{\boldsymbol{\pi} | R_m := 0\} \\
&\mathbf{forall}\, R_i\,\{\boldsymbol{\pi}\} \equiv \\
&\quad R_i := 0;\ \boldsymbol{\pi};\ R_i := (R_i{+}1) \bmod b;\ \mathbf{while}\, R_i{>}0\,\{\boldsymbol{\pi};\ R_i := (R_i{+}1) \bmod b\}
\end{aligned}$$

Fig. 6. The program setting the ancestral relation of P into the history.

6.1 Example: Ancestral relation. Suppose that the prehistory codes in the registers R_m^2 a finite binary relation $P(x,y)$ over numbers $< b$, i.e. the test in $\mathbf{if}\ \boldsymbol{h}[R_m^2]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ is satisfied iff $P(R_{m+1},R_m)$ holds. Fig. 6 shows a deterministic history setting program which sets into the history the ancestral relation P^* of P. We have $P^*(x,y)$ iff x is a P-ancestor of y (think of $P(x,y)$ as *x is a parent of y*). By switching the arguments of P the reader can visualize P as the edge relation of a directed graph and then P^* is its transitive closure.

We compute the ancestral relation $P^*(x,y) \leftrightarrow P_b(x,y)$ by the well-known algorithm of Warshall with the recurrences:

$$\begin{aligned}
P_0(x,y) &\leftrightarrow P(x,y) \\
P_{z+1}(x,y) &\leftrightarrow P_z(x,y) \vee P_z(x,z) \wedge P_z(z,y)
\end{aligned}$$

where for $0 \le z \le b$ the ternary relation $P_z(x,y)$ holds iff x is an ancestor of y entirely through elements $< z$.

The inspection of recurrences shows that the relation P_{z+1} extends the relation P_z and that P_{z+1} can be computed from P_z by adding the pairs $\langle x,y\rangle$ to the history whenever the pairs $\langle x,z\rangle$ and $\langle z,y\rangle$ are already there.

6.2 Theorem (Characterization of history setting programs). *History setting d-programs (programs) compute exactly the predicates in P (NP). Already the programs $\boldsymbol{\pi}^1$ are such.*

Proof. For the proof that the predicates of P (NP) are computed by history setting d-programs (programs) $\boldsymbol{\pi}^1$ we take a Turing machine M accepting a predicate $P(x)$ in P (NP) in polynomial time $p(b)$ with $b = |x|$. When $x \leq 2$ we compute $P(x)$ from the small predicate $P_1(x) \leftrightarrow P(x) \land x \leq 2$.

When $x > 2$ we have $b > 1$ and we compute P by a history setting d-program (program) $\boldsymbol{\pi}^1$ with $D^{\boldsymbol{\pi}} = d + a$ for two numbers d and a determined as follows.

The number d is the least one such that 2^d is greater than all labels of M. We reserve d registers R_q^d to hold the current label of M. The label $l = \sum_{i<d} l_i{\cdot}2^i$ is coded in 'binary' where the register R_{q+i} stores the digit l_i.

The number a is obtained from 2.2(1) as the least one to satisfy $p(b) + b < p(b) + b + 1 \leq b^a$. We reserve a registers R_c^a as the current simulation time of M. The time at the start of the simulation is b.

The history statements of $\boldsymbol{\pi}$ are **if** $\boldsymbol{h}[R_0^{D^{\boldsymbol{\pi}}}]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ where if $R_0^{D^{\boldsymbol{\pi}}} < b$ we access in the prehistory the input tape of M at the position R_0. If $R_0^{D^{\boldsymbol{\pi}}} \geq b$ then the registers R_0^a contain the simulation time of M and the registers R_a^d the label of M at that time. We will need the macro $R_i^d := l(R_j^a)$ computing the function $q = l(j)$ which yields the label q of M at the time j. The macro reads the label off history as follows:

$$
\begin{aligned}
R_i^d := l(R_j^a) \equiv\ & R_0^{D^{\boldsymbol{\pi}}} := R_j^a;\ R_u := 1;\\
& \textbf{while}\, R_u{>}0\,\{\textbf{if}\ \boldsymbol{h}[R_0^{D^{\boldsymbol{\pi}}}]\,\{R_u := 0|R_a^d := \text{next label}\}\};\\
& R_i^d := R_a^d\ .
\end{aligned}
$$

The currently scanned symbol on the input tape of M is held in the register R_x as 1 if the symbol is 2 and 0 if the symbol is 2. The currently scanned symbol on the work tape is held in the registers R_t^2 as $R_t^2 = 0$ if the symbol is 0, $R_t^2 = 1$ if the symbol is 1, and $R_t^2 = b + 1$ if the symbol is 2.

One step of the simulation of M by $\boldsymbol{\pi}$ inspects the current values in R_q^d, R_x, and R_t^2, determines into R_q^d the new current label of M (using

$$\textbf{rand}\,\{R_q^d := \text{the label } q_1|R_q^d := \text{the label } q_2\} \tag{1}$$

if the M's current instruction is $rand(q_1, q_2)$). The program $\boldsymbol{\pi}$ sets by the macro Set_h the new current label into history and increments the timer:

$$Set_h \equiv R_0^{D^{\boldsymbol{\pi}}} := R_q^d, R_c^a;\ \textbf{set}\ \boldsymbol{h}[R_0^{D^{\boldsymbol{\pi}}}];\ R_v, R_c^a := R_c^a{+}1\ . \tag{2}$$

The new currently scanned symbol on the input tape is then determined by the macro Set_x:

$$Set_x \equiv R_0 := Pos_x(R_c^a);\ R_1^{D^{\boldsymbol{\pi}}-1} := 0;\ \textbf{if}\ \boldsymbol{h}[R_0^{D^{\boldsymbol{\pi}}}]\,\{R_x := 1|R_x := 0\}$$

where the macro $R_i := Pos_x(R_j^a)$ implements the following recurrences for the function $Pos_x(j)$ determining the position of the input tape at the time j:

$$Pos_x(b) = 0$$
$$Pos_x(j{+}1{+}b) = \begin{cases} Pos_x(j{+}b){+}1 & \text{if } l(j{+}b) \ \textit{lbls cleft}_x \wedge Pos_x(j{+}b) < b \dot{-} 1 \\ Pos_x(j{+}b) \dot{-} 1 & \text{if } l(j{+}b) \ \textit{lbls cright}_x \wedge Pos_x(j{+}b) > 0 \\ Pos_x(j{+}b) & \text{otherwise.} \end{cases}$$

Finally, the new currently scanned symbol on the work tape is determined by the macro Set_t:

$$Set_t \equiv R_p^a := Pos_t(R_c^a);\ R_t^2 := Symb(R_c^a, R_p^a)$$

where the macro $R_i^2 := Pos_t(R_j^a)$ implements the following recurrences for the function $Pos_t(j)$ determining the position of the work tape at the time j:

$$Pos_t(b) = 0$$
$$Pos_t(j{+}1{+}b) = \begin{cases} Pos_t(j{+}b){+}1 & \text{if } l(j{+}b) \ \textit{lbls cleft}_t \\ Pos_x(j{+}b) \dot{-} 1 & \text{if } l(j{+}b) \ \textit{lbls cright}_t \wedge Pos_t(j{+}b) > 0 \\ Pos_t(j{+}b) & \text{otherwise} \end{cases}$$

and the macro $R_i^2 := Symb_t(R_{j_1}^a, R_{j_2}^a)$ implements the following recurrences for the function $Symb_t(j, w)$ determining the symbol at the position w of the working tape at the time j:

$$Symb_t(b, w) = 0$$
$$Symb_t(j{+}1{+}b, w) = \begin{cases} 1 & \text{if } Pos_t(j{+}b) = w \wedge l(j{+}b) \ \textit{lbls write}_t(1) \\ b + 1 & \text{if } Pos_t(j{+}b) = w \wedge l(j{+}b) \ \textit{lbls write}_t(2) \\ Symb_t(j{+}b, w) & \text{otherwise.} \end{cases}$$

At the beginning of simulation $\boldsymbol{\pi}$ initializes the four simulation registers by:

$$R_c^a := 0;\ R_{c+1} := 1;\ R_q^d := \text{initial label of } M;\ Set_h;\ Set_x;\ Set_t\ .$$

At the end of simulation if M accepts then $\boldsymbol{\pi}$ terminates and it goes into an infinite loop otherwise.

Vice versa, we take a predicate $P(x)$ computed by a history setting d-program (program) $\boldsymbol{\pi}^k$ with the history registers $R_i^{D^\pi}$ and by a small predicate P_1 and we construct a Turing machine M simulating $\boldsymbol{\pi}$ so the predicate $P(x)$ is in P (NP). We simulate $\boldsymbol{\pi}$ by M similarly as in the proof of Thm. 5.3 except that M stores on its work tape for $b^k = |x|$ also the Pascal-like array:

$$h : \mathbf{array}[\overbrace{0..b \dot{-} 1, \cdots, 0..b \dot{-} 1}^{D^\pi}] \ \mathbf{of\ bool}$$

holding the history of execution of $\boldsymbol{\pi}$. The array is initialized as follows

$$h[y_{D^\pi},\dots,y_1,z_k,\dots,z_1] := \begin{cases}\textbf{true} & y_{D^\pi} = \dots = y_1 = 0 \land x[[z_k,\dots,z_1]^b] = 2\\ \textbf{false} & \text{otherwise}\end{cases}$$

where the digits $x[[z_k,\dots,z_1]^b]$ are read off the input tape of M. This can be done in time $\mathcal{O}(|x|^{2\cdot(D^\pi-k)})$. M stores the digits of input x in h as prehistory and so the simulation of a history statement **if** $\boldsymbol{h}[R_i^{D^\pi}]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ leads to the reference of the history array $h[R_i^{D^\pi}]$. M performs the assignment $h[R_i^{D^\pi}] :=$ **true** into the array whenever the signal history register of $\boldsymbol{\pi}$ becomes 1.

When $P(x)$ holds then the program $\boldsymbol{\pi}$ can run for at most $L^\pi \cdot b^{M^\pi}$ steps between two settings to history and there is at most b^{D^π} such. Thus if the simulation of $\boldsymbol{\pi}$ by M does not stop in $L^\pi \cdot b^{D^\pi+M^\pi}$ steps $P(x)$ cannot hold and M should reject. For that M maintains a timer and thus it runs in polynomial time. □

6.3 Remark. The reader has probably noticed that in the example Par. 6.1 and in the proof of Thm. 6.2 we have set certain recurrences into history. Access to history is a special case of course of values recursion. Our inspiration came from the beautiful characterization of P via recursion by N. Jones [2, 3]. In order to reduce the exponential time needed for the naive implementation of recursion to polynomial time Jones uses a programming technique known as memoization which, mathematically viewed, is also a form of course of values recursion. The treatment of *Pspace* by resetting of history (see Sect. 7) as well as the shifting of history into input (see Sect. 8) are ours.

7 Characterization of Programs

Allowing the programs to reset history increases their power from polynomial time to polynomial space.

7.1 Example: Projection of a relation. Guessing of a value from among 2^b possibilities can be done by a nondeterministic history setting program or by a deterministic one which also resets history. Assume that we have a macro $R_p := P$ computing a predicate P by a history setting d-program $\boldsymbol{\pi}^2$ with history registers R_i^h where $h > 2$:

$$\boldsymbol{\pi} \equiv R_0 := P;\ \textbf{if}\, R_0 > 0\,\{R_0 := (R_0{+}1) \bmod b \mid \mathit{Fail}\}\ .$$

Further assume that $P(z)$ holds for $z > 2$ only if its input is

$$z = 1^{(b-2)\cdot b} \star y \star x$$

for some b, y, x such that $b^2 = |z|$ and $|y| = |x| = b$. We use the notation 1^a as a shorthand for $2^a - 1$. To simplify the problem let us assume that the macro for P never sets history when $R_i^h < b^2$.

We wish to find a program for the projection Q of P which for $x > 2$ satisfies:

$$Q(x) \leftrightarrow \exists b \exists y(b = |y| = |x| \wedge P(1^{(b-2)\cdot b} \star y \star x)) \ .$$

The predicate Q is in *NP* because we can just guess with random statements the b digits of y and set those which are 2 to history with $R^h_{i+1} = 1$ and R_i holding the index of the digit of y. After that we call the macro for P for verification.

If we wish to compute Q by a deterministic program then we can set and reset in history all of 2^b candidate values for y. This can be done by the following d-program:

$$\begin{aligned}
&R_g := 1; \\
&\textbf{while}\, R_g{>}0\,\{R_p := P;\ \textbf{if}\, R_p > 0\,\{R_g := 0 \mid R_g := \text{Incr. } y\}\}; \\
&\textbf{if}\, R_p > 0\,\{R_p := (R_p{+}1) \bmod b \mid \textit{Fail}\} \ .
\end{aligned}$$

Initially only the digits 2 of x are set in the prehistory and nothing in the history. This means that the candidate y is initially 1^b. If P fails then the macro $R_g :=$ Incr. y is called to increase the candidate to $y + 1$ unless we have $y = 2^b$ and then the macro returns $R_g = 0$. The macro goes from left to right through the digits of y replacing every digit 2 by the digit 1 (this is done by resetting the history for the digit). If a digit 1 is found then it is changed to 2 by setting it into the history:

$$\begin{aligned}
R_g := \text{Incr. } y \equiv R^h_i := 0;\ &R_{i+1} := 1;\ R_c := 1; \\
&\textbf{while}\, R_c > 0\,\{\textbf{if}\ \boldsymbol{h}[R^h_i]\,\{\textbf{reset}\ \boldsymbol{h}[R^h_i];\ R_c, R_i := R_i{+}1; \\
&\qquad\qquad\qquad \textbf{if}\, R_c > 0\,\{R_c := 0;\ R_g := 0 \mid R_c := 1\} \\
&\qquad\qquad \mid \textbf{set}\ \boldsymbol{h}[R^h_i];\ R_c := 0\}\} \ .
\end{aligned}$$

7.2 Theorem (Characterization of programs). *Both deterministic and arbitrary programs compute exactly the predicates in Pspace. Already the programs* $\boldsymbol{\pi}^1$ *are such.*

Proof. For the proof that the predicates of *Pspace* are computed by deterministic programs $\boldsymbol{\pi}^1$ we take a deterministic Turing machine M accepting a predicate $P(x)$ in polynomial space $p(b)$ with $b = |x|$. We construct a deterministic program $\boldsymbol{\pi}^1$ with registers R^d_q (current label of M), R^1_x (the scanned input tape symbol), R^1_i (input tape position), R^2_t (the scanned work tape symbol), and R^a_w (work tape position) where the numbers d and a are determined as in the proof of Thm. 6.2. Also the coding of R^d_q, R_x, and R^2_t is the same. Instead of the execution timer R^a_c, $\boldsymbol{\pi}$ uses the registers R^a_w to store the number $b + w$ where w is the current work tape position. The program $\boldsymbol{\pi}$ uses the history statements $\textbf{if}\ \boldsymbol{h}[R^a_0]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ where if $R^{a-1}_1 = 0$ we enquire the prehistory for the digit R_0 of M's input tape and if $R^{a-1}_1 > 0$ we enquire the history whether the work tape symbol currently written at the position $R^a_0 - b$ is 2. If the history test fails the symbol can be 0 or 1. In order to distinguish between the two, $\boldsymbol{\pi}$ maintains a counter R^a_c holding $b + i$ where i is the first position on the work tape with the symbol 0. This is tested by the macro $R_z^a := R_w^a <_* R_c^a$ and when $R_z > 0$ the symbol at the position R_0^a is 1.

The program $\boldsymbol{\pi}$ simulates M in a straightforward way except when it comes to the writing onto M's work tape. If the symbol R_t^2 to be written is 2 the program $\boldsymbol{\pi}$ sets history by: $R_0^a := R_w^a$; **set** $\boldsymbol{h}[R_0^a]$. If the symbol to be written is 1 the program $\boldsymbol{\pi}$ resets history by: $R_0^a := R_w^a$; **reset** $\boldsymbol{h}[R_0^a]$. In both cases $\boldsymbol{\pi}$ checks whether the symbol it is overwriting is 0 by testing $R_w^a < R_c^a$ and if this is the case $\boldsymbol{\pi}$ increments the counter R_c^a by one.

Vice versa, we take a predicate $P(x)$ computed by a program $\boldsymbol{\pi}^k$ with the history registers $R_i^{D^{\boldsymbol{\pi}}}$ and by a small predicate P_1 and we construct a nondeterministic Turing machine M simulating $\boldsymbol{\pi}$ in polynomial space. This is done as in the proof of Thm. 6.2 except that the history array h is set by $h[R_i^{D^{\boldsymbol{\pi}}}] :=$ **true** ($h[R_i^{D^{\boldsymbol{\pi}}}] :=$ **false**) whenever the signal history register of $\boldsymbol{\pi}$ becomes 1 with the history resetting register cleared: $R_{i+D^{\boldsymbol{\pi}}+1} = 0$ (positive: $R_{i+D^{\boldsymbol{\pi}}+1} > 0$).

When $P(x)$ holds then the program $\boldsymbol{\pi}$ can run for at most $L^{\boldsymbol{\pi}} \cdot b^{M^{\boldsymbol{\pi}}}$ steps between two settings/resettings to history and there is at most $2^{b^{D^{\boldsymbol{\pi}}}}$ such. Thus if the simulation of $\boldsymbol{\pi}$ by M does not stop in $L^{\boldsymbol{\pi}} \cdot 2^{b^{D^{\boldsymbol{\pi}}}} \cdot b^{M^{\boldsymbol{\pi}}}$ steps $P(x)$ cannot hold and M should reject. For that M maintains a timer. Both the history array h and the timer take polynomial space (with the remaining registers and counters taking logspace) and thus M runs in non-deterministic polynomial space which is the same as polynomial space *Pspace*. $\square$

7.3 Programs receiving input through registers. We can extend the definition 4.3(2) of predicates computed by basic programs $\boldsymbol{\pi}$ to predicates computed by any programs. Programs $\boldsymbol{\pi}$ receive their input through the registers, and as there is no need for the prehistory, so they are $\boldsymbol{\pi}^0$.

7.4 Theorem (Characterization of predicates defined by 4.3(2)). *Predicates defined by 4.3(2) from programs π classified as in the following table are exactly the predicates in the given complexity classes:*

programs	restrictions	compute
basic	only incr. and whiles	*Linspace*
det. simple	no hist. signal, no rand.	*Linspace*
simple	no hist. signal	*Nlinspace*
det. hist. setting	no hist. reset, no rand.	*Etime*
hist. setting	no hist. reset	*Netime*
det.	no rand.	*Espace*
any	none	*Espace*

Proof. Basic programs compute *Linspace* by Thm. 4.2 and by the Ritchie's characterization of predicates of $\mathcal{E}^2$. Simple d-programs without access to prehistory fail all their history statements **if** $\boldsymbol{h}[R_i^h]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ and so they cannot compute more than basic programs. By the same token simple programs compute *N-linspace.*

History setting d-programs (programs) run by a modification of the proof of Thm. 6.2 in time $\mathcal{O}(b^n)$ and, since $b^n \overset{2.5(1)}{\leq} 2^{2\cdot n\cdot|\max(\overline{x})|}$, they compute *Etime* (*Netime*). Programs (both with and without random statements) run by a modification of the proof of Thm. 7.2 in space $\mathcal{O}(2^{n\cdot|\max(\overline{x})|})$ and so they compute *Espace*. □

8 Converting History to Input

We can achieve the effect of memory without history statements by supplying the predicates with larger input part of which is projected away by existential quantification. *Input* programs are formed from increments, while loops, and from input statements where the latter replace the history and random statements. We will be able to characterize in Thm. 8.8 the classes $N \subseteq NL \subseteq P \subseteq NP$ by restricting in the input programs the access to the parts of input projected away. Although the characterization is intrinsic, it is not syntactical because it depends on properties of computation sequences.

$$\begin{aligned}
&\mathit{Next}^{\pi}_{x}(q,j,l) \leftrightarrow \exists l_1 \exists l_2 \exists i (l_1 = q[j]^{L^{\pi}} \wedge (\\
&\quad \mathit{Follows}^{\pi}(l_1,l,l_2) \wedge \exists b \exists y (b^{D^{\pi}} = |x| \wedge y = R^{\pi}_{b}(q,j{+}1,0) \wedge (\\
&\quad\quad \mathit{Incr}^{\pi}(l_1,i) \vee \\
&\quad\quad \mathit{Whr}^{\pi}(l_1,i) \wedge y[i]^{b} > 0 \vee \\
&\quad\quad \mathit{Inp}^{\pi}(l_1,i) \wedge x[y \div b^{i} \bmod b^{D^{\pi}}] = 2)) \vee \\
&\quad \mathit{Follows}^{\pi}(l_1,l_2,l) \wedge \exists b \exists y (b^{D^{\pi}} = |x| \wedge y = R^{\pi}_{b}(q,j{+}1,0) \wedge (\\
&\quad\quad \mathit{Whr}^{\pi}(l_1,i) \wedge y[i]^{b} = 0 \vee \\
&\quad\quad \mathit{Inp}^{\pi}(l_1,i) \wedge x[y \div b^{i} \bmod b^{D^{\pi}}] \neq 2)))) \\
&\mathit{Cseq}^{\pi}_{x}(q) \leftrightarrow \exists n (n = \mathit{Last}^{\pi}(q) \wedge q[0]^{L^{\pi}} = 0 \wedge q[n]^{L^{\pi}} = L^{\pi} \dot{-} 1 \wedge \\
&\quad\quad\quad\quad \forall j{<}n\, \mathit{Next}^{\pi}_{x}(q,j,q[j{+}1]^{L^{\pi}}))
\end{aligned}$$

Fig. 7. Computation sequences for input programs.

8.1 Input programs and their semantics. *Input* programs are generated by the grammar:

$$\begin{aligned}
&\sigma ::= R_i \mathbin{\mathbf{:=}} (R_i{+}1) \bmod b \mid \mathbf{while}\, R_i{>}0\, \{\boldsymbol{\pi}\} \mid \mathbf{if}\, x[R^m_i]{=}2\, \{\boldsymbol{\pi} \mid \boldsymbol{\pi}\} \\
&\boldsymbol{\pi} ::= \sigma \mid \sigma;\, \boldsymbol{\pi}
\end{aligned}$$

Input programs $\boldsymbol{\pi}$ with *input* statements $\mathbf{if}\, x[R^m_i]{=}2\, \{\boldsymbol{\pi} \mid \boldsymbol{\pi}\}$ must satisfy the same syntactic restrictions as the history statements. The *dimension* m of input statements of $\boldsymbol{\pi}$ is designated by the constant $D^{\boldsymbol{\pi}}$.

The semantics of input programs is given by defining $Follows^{\pi}(l, l_1, l_2)$ for input statements in the same way as for history statements and by defining the predicates $Inp^{\pi}(l, i)$ to hold if l labels $\mathbf{if}\, x[R_i^{D^{\pi}}]{=}2\,\{\pi_1 \mid \pi_2\}$. The functions $Last_b^{\pi}$ and $R_b^{\pi}(x)$ from Fig. 2 are without changes, there is no need for the predicate Hs, and in the redefined predicates $Next^{\pi}$ and $Cseq^{\pi}$ shown in Fig 7 we replace the base argument b by the input argument x (b can be recovered from x since we have $b^{D^{\pi}} = |x|$) and we omit the register arguments (since the initial value is always 0 and the final one is immaterial for the computation of predicates).

8.2 Predicates computed by input programs. Given an input program π and a small predicate P_1 we say that the unary predicate $P(x)$ is *computed by* π *and* P_1 if we have:

$$P(x) \leftrightarrow \exists b \exists q (\, b^{D^{\pi}} = |x| \land b > 1 \land Cseq_x^{\pi}(q)\,) \lor P_1(x) \;. \tag{1}$$

By a straightforward modification of the proof of Thm. 5.3 we obtain:

8.3 Theorem (Characterization of input programs). *Input programs compute exactly the predicates in L. Actually, already the input programs with* $D^{\pi} = 1$ *are such.* □

8.4 Predicates defined by projection. Let π^k where $0 < k \leq D^{\pi}$ be an input program and P_1 a small predicate. We say that the unary predicate P is defined by *projection from* π^k *and* P_1 if

$$P(x) \leftrightarrow \exists y \exists b (\, R(y \star x) \land |x| = b^k \land |y| + b^k = b^{D^{\pi}}\,)$$

and the predicate R is computed by π and P_1.

The digit j of $z = y \star x$ s.t. $b^k \leq j < b^{D^{\pi}}$, i.e. a digit of y, is called a *memory digit* of π^k. When memory digits are accessed by input statements $\mathbf{if}\, x[R_i^{D^{\pi}}]{=}2\,\{\pi_1 \mid \pi_2\}$ then we have $R_{i+k}^{D^{\pi}-k} > 0$.

8.5 References to memory digits. Let π^k be an input program. We say that π^k *references for the input* x *at the time* j *the memory digit* d *by an input statement with the label* l if we have $Refm_x^{\pi^k}(j, l, d)$ where the four place predicate designated by the symbol $Refm^{\pi^k}$ is defined by:

$$\begin{aligned} Refm_x^{\pi^k}(j, l, d) \leftrightarrow{} & \exists b \exists q \exists i (b^k = |x| \land b > 1 \land Cseq_x^{\pi}(q) \land \\ & l = q[j]^{L^{\pi}} \land Inp^{\pi}(l, i) \land d = R_b^{\pi}(q, j, 0) \div b^i \bmod b^{D^{\pi}} \land d > b^k) \;. \end{aligned}$$

8.6 Single memory references. Let π^k be an input program. We say that π^k *has single memory references* if it never accesses a memory digit more than once, i.e. if the following holds:

$$Refm_x^{\pi^k}(j_1, l_1, d) \land Refm_x^{\pi^k}(j_2, l_2, d) \rightarrow j_1 = j_2 \;.$$

8.7 Memory prediction. Let $\boldsymbol{\pi}^k$ be an input program. We call an input statement of $\boldsymbol{\pi}$ *predicting* if it has the form

$$\mathbf{if}\, x[R_i^{D^{\boldsymbol{\pi}}}]{=}2\,\{\boldsymbol{\pi}_1 \mid \mathit{Fail}\}$$

for some $\boldsymbol{\pi}_1$. Executing a predicting input statement while computing a predicate $P(x)$ means that the program must expect the referenced digit of x to be 2 for the application $P(x)$ to succeed.

We say that $\boldsymbol{\pi}^k$ *has first memory references by prediction* if the first reference to a memory digit is by a predicting input statement, i.e. if for all l whenever we have

$$\exists x \exists j \exists d (\mathit{Refm}_x^{\boldsymbol{\pi}^k}(j,l,d) \wedge \forall j_1 \forall l_1 (\mathit{Refm}_x^{\boldsymbol{\pi}^k}(j_1,l_j,d) \rightarrow j \leq j_1))$$

the input statement labelled by l is predicting.

8.8 Theorem (Characterization of predicates defined by projection). *Predicates defined by projection from input programs $\boldsymbol{\pi}^k$ with memory references restricted as in the following table are exactly in the corresponding complexity classes:*

memory references \ first memory references	*by prediction*	*unrestricted*
single	*L*	*NL*
unlimited	*P*	*NP*

Proof. We first prove that every predicate $P(x)$ computed by a Turing machine M to be in a class in the above table is defined by a projection of an input program $\boldsymbol{\pi}'^1$ with correspondingly restricted access to its memory digits.

If $P \in L$ we construct $\boldsymbol{\pi}'$ similarly as the simple d-program $\boldsymbol{\pi}$ in the proof of Thm. 5.3 except that we replace every its statement $\mathbf{if}\ \boldsymbol{h}[R_0^1]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ by the statement $\mathbf{if}\, x[R_0^1]{=}2\,\{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}$. Since $D^{\boldsymbol{\pi}} = 1$, there are no memory digits in $\boldsymbol{\pi}'$ and so its memory references are trivially by single reference by prediction.

If $P \in \mathit{NL}$ then the Turing machine M runs in time $p(b)$ for some polynomial p. We find a $h > 1$ such that $p(b)+b+1 \leq b^h$ by 2.2(1) and construct $\boldsymbol{\pi}'$ similarly as the simple program $\boldsymbol{\pi}$ in the proof of Thm. 5.3 but with $D^{\boldsymbol{\pi}'} = h$. We replace the statements $\mathbf{if}\ \boldsymbol{h}[R_0^1]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ of $\boldsymbol{\pi}$ by $R_1^{h-1} := 0;\ \mathbf{if}\, x[R_0^h]{=}2\,\{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}$. In order to make space for the new registers R_1^{h-1} we renumber all registers R_{n+1} of $\boldsymbol{\pi}$ to R_{n+h}. We replace the statements $\mathbf{rand}\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ of $\boldsymbol{\pi}$ by

$$R_0^h := R_c^h;\ R_v, R_c^h := R_c^h{+}1;\ \mathbf{if}\, x[R_0^h]{=}2\,\{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}$$

where $\boldsymbol{\pi}'$ maintains a counter R_c^h initialized to b. It should be clear that P can be defined by projection from $\boldsymbol{\pi}'^1$ and also that the memory digits of $\boldsymbol{\pi}'$ are accessed by single reference because of the counter R_c^h.

If the predicate P is in P or NP then we construct $\boldsymbol{\pi}'$ similarly as the history setting program $\boldsymbol{\pi}$ in the proof of Thm. 6.2 except that we replace every its statement $\mathbf{if}\ \boldsymbol{h}[R_0^{D^{\boldsymbol{\pi}}}]\,\{\boldsymbol{\pi}_1|\boldsymbol{\pi}_2\}$ by the statement $\mathbf{if}\, x[R_0^{D^{\boldsymbol{\pi}'}}]{=}2\,\{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}$ where

$D^{\pi'} = D^{\pi}$. We also modify the setting of history in π as follows. When the current instruction of M is not a *rand* instruction then instead of setting into history the new current label of M by 6.2(2) we predict it in the memory digits of input by:

$$R_0^{D^{\pi'}} := R_q^d, R_c^a;\ \mathbf{if}\, x[R_0^{D^{\pi'}}]{=}2\,\{R_{D^{\pi'}} := 0 \mid \mathit{Fail}\};\ R_v, R_c^a := R_c^a{+}1\ . \qquad (1)$$

When the current instruction of M is $\mathit{rand}(q_1, q_2)$ then instead of guessing it by 6.2(1) and setting it by 6.2(2) the program π' guesses q_1 or q_2 from the memory digits of input as follows:

$$R_0^a := R_c^a;\ R_a^d := \text{the label }\ q_1;$$
$$\mathbf{if}\, x[R_0^{D^{\pi'}}]{=}2\,\{R_q^d := R_a^d \mid R_a^d := \text{the label }\ q_2;\ \mathbf{if}\, x[R_0^{D^{\pi'}}]{=}2\,\{R_q^d := R_a^d \mid \mathit{Fail}\}\};$$
$$R_v, R_c^a := R_c^a{+}1\ .$$

It should be clear that P can be defined by a projection of π'^1. If M (and thus also π) is deterministic then the first memory references are by prediction (1) because of the timer R_c^a.

Vice versa, let π^k be an input program with $m = D^{\pi}$ defining a predicate P by projection. We may assume without loss of generality that if π contains input statements $\mathbf{if}\, x[R_i^m]{=}2\,\{\pi_1 \mid \pi_2\}$ then none of the registers R_{i+m}^3 is incremented in π (this can be assured by the systematic renumbering of registers in π). We construct a program π computing the same predicate as follows.

If $m = k$ then π^k has trivially single memory references by prediction and we obtain π' by replacing every input statement $\mathbf{if}\, x[R_i^m]{=}2\,\{\pi_1 \mid \pi_2\}$ of π by $\mathbf{if}\ \boldsymbol{h}[R_i^m]\,\{\pi_1|\pi_2\}$. Since the history signal register R_{i+m} is not incremented, the d program π' is simple and it computes the predicate P which is in L by Thm. 5.3.

For the remaining cases we assume $m > k$ and we leave to the reader the writing of the macro $\mathbf{if}\, R_{i+k}^{m-k} > 0\,\{\pi_1 \mid \pi_2\}$ testing for access to the memory digits.

If π^k has single memory references by prediction then we obtain π' by replacing every input statement $\mathbf{if}\, x[R_i^m]{=}2\,\{\pi_1 \mid \pi_2\}$ of π by:

$$\mathbf{if}\, R_{i+k}^{m-k} > 0\,\{\pi_1 \mid \mathbf{if}\ \boldsymbol{h}[R_i^m]\,\{\pi_1|\pi_2\}\}\ .$$

The d-program π' is simple and so it computes a predicate in L by Thm. 5.3. It computes the predicate P because if $P(x)$ holds then π succeeds for some $y \star x$. All executed input statements in π' accessing the digits of y are predicting and so they must succeed. But then also π' succeeds for x. On the other hand, if $P(x)$ does not hold then π fails for any $y \star x$. Since, there is a y making all executed input statements in π accessing digits in y true, neither π' can succeed for x.

If π has single memory references then we obtain π' by replacing every input statement $\mathbf{if}\, x[R_i^m]{=}2\,\{\pi_1 \mid \pi_2\}$ of π by:

$$\mathbf{if}\, R_{i+k}^{m-k} > 0\,\{\mathbf{rand}\,\{\pi_1|\pi_2\} \mid \mathbf{if}\ \boldsymbol{h}[R_i^m]\,\{\pi_1|\pi_2\}\}\ .$$

It should be clear that $\boldsymbol{\pi}'$ is a simple program computing P and so it is in *NL* by Thm. 5.3.

If $\boldsymbol{\pi}$ has unlimited memory references with first references by prediction then we obtain $\boldsymbol{\pi}'$ by replacing every input statement $\mathbf{if}\, x[R_i^m]{=}2\,\{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}$ of $\boldsymbol{\pi}$ by:

$$\mathbf{if}\ \boldsymbol{h}[R_i^m]\,\{\boldsymbol{\pi}_1 | \mathbf{if}\, R_{i+k}^{m-k} > 0\,\{\mathbf{set}\ \boldsymbol{h}[R_i^m];\ \boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}\}\ .$$

The d-program $\boldsymbol{\pi}'$ is history setting which sets the first memory reference to history. It computes the predicate P which is in the class *P* by Thm. 6.2.

If $\boldsymbol{\pi}$ has unrestricted memory references then we obtain $\boldsymbol{\pi}'$ by replacing every input statement $\mathbf{if}\, x[R_i^m]{=}2\,\{\boldsymbol{\pi}_1 \mid \boldsymbol{\pi}_2\}$ of $\boldsymbol{\pi}$ by:

$$\begin{array}{l}
\mathbf{if}\, R_{i+k}^{m-k} > 0\,\{R_{i+m} := 1; \\
\quad \mathbf{if}\ \boldsymbol{h}[R_i^{m+1}]\,\{R_{i+m} := 0 \mid \\
\qquad\qquad\qquad \mathbf{set}\ \boldsymbol{h}[R_i^{m+1}];\ \mathbf{rand}\,\{R_{i+m} := 0;\ \mathbf{set}\ \boldsymbol{h}[R_i^{m+1}] | R_{i+m} := 0\}\}; \\
\mid R_{i+m} := 0\}; \\
\mathbf{if}\ \boldsymbol{h}[R_i^{m+1}]\,\{\boldsymbol{\pi}_1 | \boldsymbol{\pi}_2\}\ .
\end{array}$$

Notice that we have $D^{\boldsymbol{\pi}'} = m + 1$ where we use the setting $R_{i+m} = 1$ in the history accesses to the memory as a switch that we have already referenced the history location R_i^m. During the first reference to the location the program $\boldsymbol{\pi}'$ randomly decides whether to set it to the history or not. All subsequent references to the location enquire this decision. The program $\boldsymbol{\pi}'$ is history setting, it computes P, and so it is in the class *NP* by Thm. 6.2. □

References

1. A. Grzegorczyk. *Some classes of recursive functions.* Rozprawy Matem. vol 4, Warsaw 1953.
2. N. D. Jones. *Computability and complexity from a programming perspective.* MIT Press. 1997
3. N. D. Jones. *LOGSPACE and PTIME characterized by programming languages.* Theor. Comp. Sci. 228, pp. 151-174. 1999.
4. R. W. Ritchie. *Classes of predictably computable functions.* Trans. Amer. Math. Soc., 106, pp. 139-173. 1963.
5. H. E. Rose. *Subrecursion: functions and hierarchies.* Oxford Logic Guides. Clarendon Press Oxford 1984.
6. J. C. Warkentin. *Small classes of recursive functions and relations.* Ph.D. dissertation CSRR2052, Dept. of Appl. Analysis and Comp. Sci., University of Waterloo, 1971.
7. J. C. Warkentin and P. C. Fischer. *Predecessor machines and regressing functions.* STOC 1972.