comput. complex. **14** (2005), 72 – 88
1016-3328/05/010072–17
DOI 10.1007/s00037-005-0191-0

computational complexity

NEAT FUNCTION ALGEBRAIC CHARACTERIZATIONS OF LOGSPACE AND LINSPACE

LARS KRISTIANSEN

Abstract. We characterize complexity classes by function algebras that neither contain bounds nor any kind of variable segregation. The class of languages decidable in logarithmic space is characterized by the closure of a neat class of initial functions (projections and constants) under composition and simultaneous recursion on notation. We give a similar characterization of the class of number-theoretic 0-1 valued functions computable in linear space using simultaneous recursion on natural numbers in place of simultaneous recursion on notation.

Keywords. Function algebras, recursion schemes, LOGSPACE, LINSPACE.

Subject classification. 68Q15, 68Q05, 03D10, 03D20.

1. Introduction

Complexity classes defined by Turing machines, like e.g. LOGSPACE, LINSPACE, P, PSPACE, can be characterized by function algebras, that is, a complexity class can be characterized by a closure of some initial function under certain composition and recursion schemes. Cobham's characterization of the polynomial time computable functions and Ritchie's characterization of the linear space computable functions are typical examples (Cobham 1965; Ritchie 1963). Until around 1990 every such characterization was based on some version of *bounded* recursion. Then it was discovered that complexity classes can also be characterized by using *unbounded* recursion schemes which *distinguish between variables* as to their position in a function (e.g. Simmons 1988; Leivant 1991, 1993; Bellantoni & Cook 1992). The Bellantoni–Cook characterization of the polynomial time computable functions is a typical example. The segregation techniques which go back to these authors are often called *ramification* or *tiering* techniques in the literature. See Clote (1999) for more historical details.

In this paper we characterize complexity classes by function algebras that *contain neither bounds nor any kind of variable segregation.*

We prove that the class of languages decidable in logarithmic space is characterized by the closure of a neat class of initial functions (projections and constants) under composition and simultaneous recursion on notation, i.e. the scheme

$$\begin{aligned} f_i(\vec{x},\varepsilon) &= g_i(\vec{x}),\\ f_i(\vec{x},0y) &= h_i^0(\vec{x},y,f_1(\vec{x},y),\dots,f_k(\vec{x},y)),\\ f_i(\vec{x},1y) &= h_i^1(\vec{x},y,f_1(\vec{x},y),\dots,f_k(\vec{x},y)), \end{aligned}$$

for $i = 1,\dots,k$. If f is generated by this algebra, then there exists a fixed $m \in \mathbb{N}$ such that if $f(w_1,\dots,w_n) = v$ and $|v| \geq m$, then v equals an end segment of some w_i. This allows a Turing machine working in logarithmic space to keep track of $f(w_1,\dots,w_n)$ by keeping track of the length of the end segments of the inputs $w_1,\dots,w_n$. Given this observation, it is easy to see that a language decided by a function in the algebra can also be decided by a Turing machine working in logarithmic space. It is harder to see that every language decided by a Turing machine working in logarithmic space can be decided by a function in the algebra. To prove this, we make a detour via a computation in a specially tailored imperative programming language.

Further, we characterize the class of number-theoretic 0-1 valued functions computable in linear space as the 0-1 valued functions in the closure of a neat class on initial functions (projections and constants) under composition and simultaneous recursion, i.e. the scheme

$$f_i(\vec{x},0) = g_i(\vec{x}), \qquad f_i(\vec{x},y+1) = h_i(\vec{x},y,f_1(\vec{x},y),\dots,f_k(\vec{x},y)),$$

for $i = 1,\dots,k$. For every function f in the algebra we have

$$f(\vec{x}) \leq \max(\vec{x},m) \tag{1.1}$$

for some fixed $m \in \mathbb{N}$. The set of linear space computable functions with the property (1.1) is closed under composition and simultaneous recursion, and hence it follows easily that every function in the algebra can be computed by a Turing machine working within the required space restrictions. Again the inclusion the other way around is harder to prove, and again the proof makes a detour via a computation in an imperative programming language.

Finally, the reader should note that we do not characterize classes of functions computable within certain resource constraints, e.g. we do not characterize the class of number-theoretic functions computable in linear space, but the class of *0-1 valued* number-theoretic functions computable in linear space. To characterize complexity-theoretic function classes by function algebras, some kind of variable segregation, or explicit bounds, seem necessary.

2. Preliminaries

We assume the reader is familiar with basic complexity theory and some recursion theory, e.g. Turing machines, complexity classes, etc. See e.g. Odifreddi (1999) or Clote (1999). We also assume familiarity with imperative programming languages. We will use informal Hoare-like sentences to specify or reason about imperative programs, that is, we will use the notation $\{A\}\,\mathtt{P}\,\{B\}$, the meaning being that if the condition given by the sentence A is fulfilled before $\mathtt{P}$ is executed, then the condition given by the sentence B is fulfilled after the execution of $\mathtt{P}$. For example, $\{\mathtt{X}=x,\mathtt{Y}=y\}\mathtt{P}\{\mathtt{X}=x',\mathtt{Y}=y'\}$ reads as *if the values x and y are held by the variables $\mathtt{X}$ and $\mathtt{Y}$, respectively, before the execution of $\mathtt{P}$, then the values x' and y' are held by $\mathtt{X}$ and $\mathtt{Y}$ after the execution of $\mathtt{P}$.* (We use typewriter style uppercase letters, with or without subscripts and superscripts, to denote program variables.) Another typical example is $\{\vec{\mathtt{X}}=\vec{x}\}\,\mathtt{P}\,\{\mathtt{Y}=f(\vec{x})\}$ meaning that if the values $\vec{x}$ are held by $\vec{\mathtt{X}}$, respectively, before the execution of $\mathtt{P}$, then the value held by $\mathtt{Y}$ after the execution of $\mathtt{P}$ equals $f(\vec{x})$. Here $\vec{x}$ (resp. $\vec{\mathtt{X}}$) abbreviates as usual the list $x_1,\ldots,x_n$ (resp. $\mathtt{X}_1,\ldots,\mathtt{X}_n$) where n is some arbitrary but fixed natural number, and f is an n-ary function. Whenever convenient $\vec{\mathtt{X}}$ will also denote a *set* of program variables $\{\mathtt{X}_1,\ldots,\mathtt{X}_n\}$. We will also use vector notation for functions, and e.g. $\vec{\mathtt{X}}=\vec{f}(y)$ abbreviates the list $\mathtt{X}_1=f_1(y),\ldots,\mathtt{X}_n=f_n(y)$ where n is some fixed number. $\mathcal{V}(\mathtt{P})$ denotes the set of variables occurring in a program $\mathtt{P}$. A program $\mathtt{P}$ computes a function f when $\{\vec{\mathtt{X}}=\vec{x}\}\mathtt{P}\{\mathtt{Y}=f(\vec{x})\}$ for some $\vec{\mathtt{X}},\mathtt{Y}\in\mathcal{V}(\mathtt{P})$. (If $\vec{\mathtt{Z}}=\mathcal{V}(\mathtt{P})\setminus\vec{\mathtt{X}}$, we expect $\{\vec{\mathtt{X}}=\vec{x},\vec{\mathtt{Z}}=\vec{z}\}\mathtt{P}\{\mathtt{Y}=f(\vec{x})\}$ to hold for any $\vec{z}$.) When we construct programs, we occasionally need what we call *fresh* variables. That a variable is *fresh* simply means that the variable is not used elsewhere. Occasionally, we will also call variables *registers*.

We will use some notation and terminology from Clote (1999). An *operator*, here also called *(definition) scheme*, is a mapping from functions to functions. If $\mathcal{X}$ is a set of functions, and OP is a collection of operators, then $[\mathcal{X};\mathrm{OP}]$ denotes the smallest set of functions containing $\mathcal{X}$ and closed under the operations of OP. The set $[\mathcal{X};\mathrm{OP}]$ is called a *function algebra.* COMP denotes the definition scheme called *composition*, i.e. the scheme $f(\vec{x})=h(g_1(\vec{x}),\ldots,g_m(\vec{x}))$ where $m\geq 0$. BR denotes the *bounded recursion* scheme, i.e. the scheme

$$f(\vec{x},0)=g(\vec{x}),\qquad f(\vec{x},y+1)=h(\vec{x},y,f(\vec{x},y)),\qquad f(\vec{x},y)\leq j(\vec{x},y).$$

SIM and SIMN denote, respectively, the schemes for *simultaneous recursion* and *simultaneous recursion on notation* given in the introductory section. (Note that SIM is a classical recursion scheme studied intensively in the literature over the years, e.g. in Hilbert & Bernays 1934 and Péter 1957. The scheme SIMN is SIM generalized to recursion on notation.) Let I_i^n denote the projection function $I_i^n(x_1,\dots,x_i,\dots,x_n) = x_i$ where i,n are fixed numbers such that $1 \leq i \leq n$. Let I denote the set of all such projection functions. Let C_k denote the 0-ary constant function $C_k = k$ where $k \in \mathbb{N}$ ($k \in \mathbb{W}$). Let $C_{\mathbb{N}}$ ($C_{\mathbb{W}}$) denote the set of all such constant functions. If $\mathcal{X}$ is a class of number-theoretic functions, then $\mathcal{X}_\star$ denotes the 0-1 valued functions in $\mathcal{X}$, i.e. $\mathcal{X}_\star = \{f \mid f \in \mathcal{X}$ and $\mathrm{ran}(f) = \{0,1\}\}$.

3. LINSPACE

Definition 3.1. We define LINSPACE as the class of number-theoretic 0-1 valued functions computable by a Turing machine working in linear space.

We now define the imperative programming languages `LOOP` and `LOOP`$^-$.

The syntax of `LOOP` is as follows. The language `LOOP` has an infinite supply of variables. (i) `suc(X)` is a program if `X` is a variable; (ii) `pred(X)` is a program if `X` is a variable; (iii) `Y:= X` is a program if `X` and `Y` are variables; (iv) `Y:= k` is a program if `Y` is a variable and `k` denotes a value in $\mathbb{N}$; (v) $\mathtt{P}_1$`;` $\mathtt{P}_2$ is a program if $\mathtt{P}_1$ and $\mathtt{P}_2$ are programs; (vi) `loop X [P]` is a program if `P` is a program, `X` is a variable, and `X` $\notin \mathcal{V}($`P`$)$; (vii) nothing else is a program.

The semantics of `LOOP` is as expected from the syntax. Each variable holds a natural number, i.e. a value in $\mathbb{N}$. The instruction `suc(X)` increases the number held by `X` by 1. The instruction `pred(X)` decreases the number held by `X` by 1 (if possible; otherwise the instruction does nothing). The instructions `Y:= X` and `Y:= k` are ordinary assignment operations. (Note that for each $k \in \mathbb{N}$ we have an operation `Y:= k` where `k` denotes k.) To execute the program $\mathtt{P}_1$`;` $\mathtt{P}_2$ we first execute the program $\mathtt{P}_1$, then the program $\mathtt{P}_2$. To execute the program `loop X [P]`, we execute the subprogram `P`, x times in a row, where x is the value in the register `X` when the execution starts. The value held by `X` will not be modified during the execution. (Note that `X` does not occur in `P`.)

A `LOOP` program `P` is a `LOOP`$^-$ program if `P` does not have a subprogram of the form `suc(X)`. We define $\mathcal{L}^-$ as the class of functions computable by a `LOOP`$^-$ program.

A programming language similar to `LOOP` is studied in Meyer & Ritchie (1967).

Lemma 3.2. *Let* `P` *be a* `LOOP` *program where* $\mathcal{V}(\mathtt{P}) = \vec{\mathtt{X}}$. *Further, assume that no register during an execution of* `P` *on input* $\vec{\mathtt{X}} = \vec{x}$ *exceeds some fixed* $m \in \mathbb{N}$.

Then there exists a LOOP$^-$ *program* Q *such that*

$$\{\vec{X} = \vec{x}\}P\{\vec{X} = \vec{x}'\} \;\Leftrightarrow\; \{\vec{X} = \vec{x}, M = m\}Q\{\vec{X} = \vec{x}'\}.$$

Proof. Note that if $m > x$, then $m \dot{-} ((m \dot{-} x) \dot{-} 1) = x + 1$. Thus, let M and Y be fresh variables, and let Q be P where each subprogram of the form `suc(X)` is replaced by the subprogram

```
Y:= M; loop X [pred(Y)]; pred(Y); X:= M; loop Y [pred(X)].
```

□

Lemma 3.3. *There exists a* LOOP$^-$ *program* P *such that*

$$\{X = x, Y = y\}P\{Z = \max(x, y)\}.$$

Moreover, no register during the execution of P *on input* $X = x, Y = y$ *exceeds* $\max(x, y, 1)$.

Proof. Let P be the program

```
U:= X; loop Y [pred(U)];
V:= 1; loop U [pred(V)]; Z:= X; loop V [Z:= Y]
```

□

Lemma 3.4. *Let* P *be a* LOOP *program and assume* $\mathcal{V}(P) = \vec{X}$. *Further, assume that no register during an execution of* P *on input* $\vec{X} = \vec{x}$ *exceeds* $\max(\vec{x}, m)^n$ *for some fixed* $m, n \in \mathbb{N}$. *Then there exists a* LOOP *program* Q *and a fixed* $b \in \mathbb{N}$ *such that (i) no register during an execution of* Q *on input* $\vec{Y} = \vec{y}$ *exceeds* $\max(\vec{y}, b)$ *(where* $\mathcal{V}(P) \subseteq \mathcal{V}(Q) = \vec{Y}$*) and (ii) if* P *computes a* 0-1 *valued function* f*, then* Q *computes* f.

Proof. Let P be as in the lemma. Let $k \in \mathbb{N}$ be the largest number such that P has a subprogram of the form `X:= k` where `k` denotes k, and let $b = \max(\vec{x}, m, k, 1) + 1$. Suppose a register holds the number a during the execution of P on input $\vec{X} = \vec{x}$. Then there exist $a_0, \ldots, a_{n-1} < b$ such that $a = a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \ldots + a_0 b^0$, i.e. a can be represented by n digits in base b. We will construct the program Q in the lemma such that each variable $X \in \mathcal{V}(P)$ is represented in base b by the variables $X^{n-1}, \ldots, X^0 \in \mathcal{V}(Q)$.

Given that the number $b - 1$ is stored in a register M, we can construct a LOOP program `suc`$(X^{n-1} \ldots X^0)$ (resp. `pred`$(X^{n-1} \ldots X^0)$) simulating the operation `suc(X)` (resp. `pred(X)`) in base b such that no variable will hold a number $\geq b$ during the operation. We leave the details to the reader.

Let Y be a fresh variable, and let `if` $X^{n-1} \ldots X^0$ `> 0 [R]` $\equiv$

```
Y:= 0; loop X^{n-1} [Y:= 1]; ... loop X^0 [Y:= 1]; loop Y [R].
```

So the program $\texttt{if X}^{n-1}\ldots\texttt{X}^0\texttt{ > 0 [R]}$ executes $\texttt{R}$ if the number represented in base b by the contents of $\texttt{X}^{n-1},\ldots,\texttt{X}^0$ does not equal 0; otherwise the program does not modify any of the variables occurring in $\texttt{R}$.

We will now define the program transformation τ recursively over the build-up of a $\texttt{LOOP}$ program. Fix variables $\texttt{M}_0,\ldots,\texttt{M}_{n-1}\notin\mathcal{V}(\texttt{P})$ and let

1. $\tau(\texttt{suc(X)})\equiv\texttt{suc(X}^{n-1}\ldots\texttt{X}^0\texttt{)}$
2. $\tau(\texttt{pred(X)})\equiv\texttt{pred(X}^{n-1}\ldots\texttt{X}^0\texttt{)}$
3. $\tau(\texttt{Y:= X})\equiv\texttt{Y}^{n-1}\texttt{:= X}^{n-1}\texttt{; }\ldots\texttt{ Y}^1\texttt{:= X}^1\texttt{; Y}^0\texttt{:= X}^0$
4. $\tau(\texttt{X:= k})\equiv\texttt{X}^{n-1}\texttt{:= 0; }\ldots\texttt{ X}^1\texttt{:= 0; X}^0\texttt{:= k}$
5. $\tau(\texttt{Q; R})\equiv\tau(\texttt{Q})\texttt{; }\tau(\texttt{R})$
6. $\tau(\texttt{loop X [R]})\equiv$

   $\tau(\texttt{Z:= X})$;
   $\texttt{loop M}_0\texttt{ [}\ldots\texttt{loop M}_{n-1}\texttt{ [if Z}^{n-1}\ldots\texttt{Z}^0\texttt{ > 0 [}\tau(\texttt{R})\texttt{]; }\tau(\texttt{pred(Z)})\texttt{]}\ldots\texttt{]}$

   where $\texttt{Z}$ is a fresh variable.

This completes the definition of τ. Note that the number denoted by $\texttt{k}$ in 4 is strictly less than the base b.

Now assume $\mathcal{V}(\texttt{P})=\vec{\texttt{X}}=\texttt{X}_1,\ldots,\texttt{X}_p$ where $\texttt{P}$ is the program in the lemma. For $i=1,\ldots,p$ let $\texttt{R}_i\equiv\texttt{X}_i^{n-1}\texttt{:= 0; }\ldots\texttt{X}_i^1\texttt{:= 0; X}_i^0\texttt{:= X}_i$. By Lemma 3.3 there exists a $\texttt{LOOP}^-$ program $\texttt{Q}_0$ such that (i) $\{\vec{\texttt{X}}=\vec{x}\}\texttt{Q}_0\{\vec{\texttt{X}}=\vec{x},\texttt{M}=\max(\vec{x},b-1)\}$ where $\texttt{M}\notin\vec{\texttt{X}}$ and (ii) during the execution of $\texttt{Q}_0$ on input $\vec{\texttt{X}}=\vec{x}$ no register exceeds $\max(\vec{x},b-1)$.

Finally, let $\texttt{Q}\equiv\texttt{R}_1\texttt{; }\ldots\texttt{R}_p\texttt{; Q}_0\texttt{; M}_0\texttt{:= M; }\ldots\texttt{M}_{n-1}\texttt{:= M; }\tau(\texttt{P})$. Then (i) no variable in $\mathcal{V}(\texttt{Q})=\vec{\texttt{Y}}$ will exceed $\max(\vec{y},b-1)$ during an execution of $\texttt{Q}$ on input $\vec{\texttt{Y}}=\vec{y}$, and (ii) if $\texttt{P}$ computes a 0-1 valued function f and leaves the result in the register $\texttt{Z}$, i.e. if $\{\vec{\texttt{X}}=\vec{x}\}\texttt{P}\{\texttt{Z}=f(\vec{x})\}$, then $\texttt{Q}$ will also compute f and leave the result in the register $\texttt{Z}^0$, i.e. $\{\vec{\texttt{X}}=\vec{x}\}\texttt{Q}\{\texttt{Z}^0=f(\vec{x})\}$. This completes the proof. □

Lemma 3.5. $\mathcal{L}^-\subseteq[I,C_{\mathbb{N}};\textsc{comp},\textsc{sim}]$.

Proof. Let $\texttt{P}$ be a $\texttt{LOOP}^-$ program and assume $\mathcal{V}(\texttt{P})=\vec{\texttt{X}}$. We will prove by induction on the build-up of $\texttt{P}$ that there exist $\vec{f}\in[I,C_{\mathbb{N}};\textsc{comp},\textsc{sim}]$ such that $\{\vec{\texttt{X}}=\vec{x}\}\texttt{P}\{\vec{\texttt{X}}=\vec{f}(\vec{x})\}$.

Assume $\mathtt{P} \equiv \mathtt{X:= Y}$. Then we have $\{\mathtt{X} = x, \mathtt{Y} = y\}\mathtt{P}\{\mathtt{X} = I^2_2(x,y), \mathtt{Y} = I^2_2(x,y)\}$ and I^2_2 is one of the initial functions in $[I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$.

Assume $\mathtt{P} \equiv \mathtt{X:= k}$ where $\mathtt{k}$ denotes the element $k \in \mathbb{N}$. Let $f(x) = C_k$. Then we have $\{\mathtt{X} = x\}\mathtt{P}\{\mathtt{X} = f(x)\}$ and $f \in [I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$.

Assume $\mathtt{P} \equiv \mathtt{pred(X)}$. Let $f(0) = C_0$ and $f(y+1) = I^2_1(y, f(y))$. Then we have $\{\mathtt{X} = x\}\mathtt{P}\{\mathtt{X} = f(x)\}$ and $f \in [I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$.

Assume $\mathtt{P} \equiv \mathtt{Q;R}$ where $\mathcal{V}(\mathtt{Q}) = \vec{\mathtt{Y}}$ and $\mathcal{V}(\mathtt{R}) = \vec{\mathtt{Z}}$. The induction hypothesis yields functions $\vec{g}, \vec{h} \in [I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$ such that $\{\vec{\mathtt{Y}} = \vec{y}\}\mathtt{Q}\{\vec{\mathtt{Y}} = \vec{g}(\vec{y})\}$ and $\{\vec{\mathtt{Z}} = \vec{z}\}\mathtt{R}\{\vec{\mathtt{Z}} = \vec{h}(\vec{z})\}$. Use the functions $\vec{g}, \vec{h}$, projection functions and the composition scheme to generate functions $\vec{f}$ such that $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{f}(\vec{x})\}$ where $\vec{\mathtt{X}} = \mathcal{V}(\mathtt{Q}) \cup \mathcal{V}(\mathtt{R})$. (The details are straightforward, but tedious. Note that for any function $f \in [I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$ there exists a function $f' \in [I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$ such that $f'(\vec{x}, \vec{y}) = f(\vec{x})$ for any $\vec{y}$.)

Assume $\mathtt{P} \equiv \mathtt{loop\ X\ [Q]}$ where $\mathtt{X} \notin \mathcal{V}(\mathtt{Q}) = \{\mathtt{Y}_1, \dots, \mathtt{Y}_k\}$. The induction hypothesis yields functions $\vec{g} \in [I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$ such that $\{\vec{\mathtt{Y}} = \vec{y}\}\mathtt{Q}\{\vec{\mathtt{Y}} = \vec{g}(\vec{y})\}$. Let

- $g'_i(y, \vec{x}) = g_i(I^{k+1}_2(y, \vec{x}), \dots, I^{k+1}_{k+1}(y, \vec{x}))$ (so $g'_i(y, \vec{x}) = g_i(\vec{x})$ for any y),
- $f_i(\vec{y}, 0) = I^k_i(\vec{y})$ and $f_i(\vec{y}, z+1) = g'_i(z, f_1(\vec{y}, z), \dots, f_k(\vec{y}, z))$,

for $i = 1, \dots, k$. Then we have $\{\vec{\mathtt{Y}} = \vec{y}, \mathtt{X} = x\}\mathtt{P}\{\vec{\mathtt{Y}} = \vec{f}(\vec{y}, x), \mathtt{X} = I^{k+1}_{k+1}(\vec{y}, x)\}$. Furthermore, $\vec{f}$ and I^{k+1}_{k+1} are functions in $[I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]$. $\square$

Lemma 3.6. *Let $\mathcal{F}$ be the set of all number-theoretic functions f such that (i) $f(\vec{x}) \leq \max(\vec{x}, m)$ for some fixed $m \in \mathbb{N}$, and (ii) f is computed by a Turing machine working in linear space. Then $\mathcal{F}$ is closed under composition* (comp) *and simultaneous recursion* (sim)*.*

Proof. We leave this (easy) proof to the reader. $\square$

Theorem 3.7. $\textsc{linspace} = [I, C_{\mathbb{N}}; \textsc{comp}, \textsc{sim}]_\star$.

Proof. Assume $f \in \textsc{linspace}$. The class of number-theoretic functions computable by a Turing machine working in linear space equals the Grzegorczyk class $\mathcal{E}^2 \overset{\text{def}}{=} [0, I, s, +, x^2+2; \textsc{comp}, \textsc{br}]$. (This is a well-known result proved by Ritchie 1963.) Thus, $f \in \mathcal{E}^2$. It is easy to prove by induction on the build-up of f from functions in the algebra $[0, I, s, +, x^2+2; \textsc{comp}, \textsc{br}]$ that there exist fixed $m, n \in \mathbb{N}$ and a $\mathtt{LOOP}$ program $\mathtt{P}$ computing f such that no register in $\mathcal{V}(\mathtt{P})$ will exceed $\max(\vec{x}, m)^n$ during the computation of $f(\vec{x})$. Thus, by Lemma 3.4, there will be a fixed $k \in \mathbb{N}$ and a `LOOP` program `P` computing f such that no register in $\mathcal{V}(\mathtt{P})$ will exceed $\max(\vec{x}, k)$ during the computation of $f(\vec{x})$. Thus, it follows from Lemmas 3.2 and 3.3 that f can be computed by a $\mathtt{LOOP}^-$ program. Thus, $f \in \mathcal{L}^-_\star$. By Lemma 3.5, we have $f \in [I, C_\mathbb{N}; \textsc{comp}, \textsc{sim}]_\star$. This proves $\textsc{linspace} \subseteq [I, C_\mathbb{N}; \textsc{comp}, \textsc{sim}]_\star$.

To prove the inclusion the other way around, we note that for any $f \in [I, C_\mathbb{N}; \textsc{comp}, \textsc{sim}]$ there exists a fixed $m \in \mathbb{N}$ such that $f(\vec{x}) \leq \max(\vec{x}, m)$. Hence, we have $[I, C_\mathbb{N}; \textsc{comp}, \textsc{sim}]_\star \subseteq \textsc{linspace}$ by Lemma 3.6. $\square$

Corollary 3.8. *A number-theoretic* 0-1 *valued function is computable by a Turing machine working in linear space if and only if the function can be defined from constant functions ($C_\mathbb{N}$) and projection functions (I) by composition and simultaneous recursion.*

4. logspace

Definition 4.1. A *word* is a string of bits, i.e. a string over the bit alphabet $\{0, 1\}$. Let ε denote the empty word. Let $\mathbb{W}$ denote the set of words. A word y is an *end segment* of a word x when there exists a word z such that $zy = x$. A *language* is a subset of $\mathbb{W}$. A language A is in logspace when A can be decided by a Turing machine working in logarithmic space. (A Turing machine M decides a language A when M on input x halts in a distinguished accept state q_A if $x \in A$, and in a distinguished reject state q_R if $x \notin A$.) A *function* $f : \mathbb{W} \to \mathbb{W}$ *decides* a language A when $f(x) = \varepsilon \Leftrightarrow x \in A$. A *program decides* a language A when it computes a function which decides A.

We shall define the imperative programming language `CLIP`.

The syntax of `CLIP` is as follows. The language `CLIP` has an infinite supply of variables. (i) `clip(X)` is a program if `X` is a variable; (ii) `Y:= X` is a program if `X` and `Y` are variables; (iii) `Y:= w` is a program if `Y` is a variable and `w` denotes a word; (iv) $\mathtt{P}_1$`;` $\mathtt{P}_2$ is a program if $\mathtt{P}_1$ and $\mathtt{P}_2$ are programs; (v) `if left(X)=0 [P]` is a program if `P` is a program and `X` is a variable; (vi) `if left(X)=1 [P]` is a program if `P` is a program and `X` is a variable; (vii) `foreach X [P]` is a program if `P` is a program, `X` is a variable, and $\mathtt{X} \notin \mathcal{V}(\mathtt{P})$; (viii) nothing else is a program.

The semantics of `CLIP` is as expected from the syntax. Each variable holds a value in $\mathbb{W}$. The instructions `Y:= X` and `Y:= w` are ordinary assignment operations. (Note that for each $w \in \mathbb{W}$ we have an operation `Y:= w` where `w` denotes w.) The instruction `clip(X)` removes the leftmost bit from the word held by `X` (if possible; otherwise the instruction does nothing). To execute the program $\mathtt{P}_1$;$\mathtt{P}_2$, we first execute the program $\mathtt{P}_1$, then we execute the program $\mathtt{P}_2$. To execute the program `if left(X)=0 [P]`, we execute `P` if $\mathtt{X} = 0x$ for some

$x \in \mathbb{W}$; otherwise we do nothing. To execute the program `if left(X)=1 [P]`, we execute `P` if $\texttt{X} = 1x$ for some $x \in \mathbb{W}$; otherwise we do nothing. To execute the program `foreach X [P]`, we execute the subprogram `P`, n times in a row, where n is the length of the word held by `X`. The value held by `X` will not be modified during the execution. (Note that `X` does not occur in `P`.)

Lemma 4.2. *Every language in* LOGSPACE *can be decided by a* `CLIP` *program.*

Proof. A 2-tape Turing machine is specified by $(Q, \Sigma, \Gamma, \delta, q_0)$ where Q is a finite set of states containing the accept state q_A and the reject state q_R; Σ (resp. Γ) is a finite read-only input (resp. read-write work) tape alphabet not containing the blank symbol B; δ is the transition function and maps $(Q \setminus \{q_A, q_R\}) \times (\Sigma \cup \{B\}) \times (\Gamma \cup \{B\})$ into $Q \times (\Gamma \cup \{B\}) \times \{-1, 0, 1\} \times \{-1, 0, 1\}$.

Let M be a 2-tape Turing machine and assume that (i) M terminates (in state q_A or q_R) on any input, (ii) M's input alphabet Σ equals $\{0, 1\}$ and (iii) the number of cells visited on M's work tape is bounded by $k_0 \log_2(|x| + 2)$ for some fixed $k_0 \in \mathbb{N}$ ($|x|$ denotes the length of the input). It will be sufficient to prove that there exists a `CLIP` program `P` such that

$$\{\texttt{X} = x\}\texttt{P}\{\texttt{Y} = \varepsilon\} \;\Leftrightarrow\; M \text{ accepts } x.$$

The proof is long. First we will simulate M by a program $\texttt{P}_0$ written in an expressive informal imperative programming language. Then we will transform $\texttt{P}_0$ into a program $\texttt{P}_1$ written in a mixture of the languages `CLIP` and `LOOP`. Thereafter we will do several transformations of $\texttt{P}_1$ and achieve a program $\texttt{P}_2$ written in a mixture of `CLIP` and $\texttt{LOOP}^-$ code. Finally, we will transform $\texttt{P}_2$ into the desired `CLIP` program `P`.

In the informal high level language, we represent the work tape by two stacks, `leftW` and `rightW` over $\Gamma \cup \{B\}$. The tape configuration $\alpha\underline{a}\beta$ is represented by $\texttt{leftW} = \alpha^R$ and $\texttt{rightW} = a\beta$ where α^R denotes the sequence α reversed. (Convention: When a stack stores a sequence of symbols α, the leftmost symbol of α should be at the top of the stack and the rightmost at the bottom.) Thus we can simulate the head's movement on the input tape smoothly by a few standard operations on stacks. The input tape is represented similarly by the stacks `leftI` and `rightI` over $\Sigma \cup \{B\}$. The transition function δ can be viewed as a finite table $T_1, \ldots, T_\ell$. The ith entry in the table T_i has the form $q, a, b, (q', b', v, w)$ where $\delta(q, a, b) = (q', b', v, w)$. For each entry T_i we construct the program $\texttt{Q}_i \equiv$

```
if state=$q$ and top(rightI)=$a$ and top(rightW)=$b$ then
   begin state:=$q'$; scW:=$b'$; moveI($v$); moveW($w$) end
```

where the subprogram `moveI(`v`)` (resp. `moveW(`w`)`) simulates the movement of the scanning head on the input (resp. working) tape. Let `init` be a procedure initiating the stacks properly. Then the following program $\mathtt{P}_0$ simulates M. $\mathtt{P}_0 \equiv$

```
init; state:=$q_0$;
while not (state=$q_A$ or state=$q_R$) do begin Q$_1$; ...;Q$_k$ end.
```

Our next task is to implement $\mathtt{P}_0$ in a mixture of `CLIP` and `LOOP` code. Thus, the language has two types of variables, $\mathbb{W}$-variables and $\mathbb{N}$-variables. We represent M's input tape by $\mathbb{W}$-variables, and we implement the procedure `moveI` in pure `CLIP` code. To see that this is possible, note the following (in the indicated order). (i) M's input alphabet is $\{0,1\}$, and we can, without loss of generality, assume that M never scans any other blank cells than the blank cell immediately to the left of the input and the blank cell immediately to the right of the input. Hence, the input tape configuration $\alpha\underline{b}\beta$ can be represented as a difference list by two $\mathbb{W}$-variables $\mathtt{S} = b\beta$ and $\mathtt{M} = \alpha b\beta$. In addition to the difference list we need two "status" $\mathbb{W}$-variables `T` and `U`; the variable `T` holds the word `0` if a blank cell is scanned, otherwise `T` holds the word `1`; the variable `U` holds the word `0` if the blank cell to the right of the input is scanned, otherwise `U` holds the word `1`. (ii) To simulate the transformation from the tape configuration $\alpha\underline{a}b\beta$ to the tape configuration $\alpha a\underline{b}\beta$, it is sufficient to execute the operation `clip(S)`. (iii) To simulate the transformation from the tape configuration $\alpha a\underline{b}\beta$ to the tape configuration $\alpha\underline{a}b\beta$, we execute the `CLIP` code

```
A:= M; foreach S [clip(A)]; clip(A); S:= M; foreach A [clip(S)]
```

where `A` is fresh. From (i), (ii), and (iii) we conclude that the input tape can be represented by $\mathbb{W}$-variables and that the configuration of the input tape can be updated by pure `CLIP` code. The remaining parts of the program $\mathtt{Q}_1$; …;$\mathtt{Q}_k$, i.e. the body of the `while` loop in $\mathtt{P}_0$, will be implemented in `LOOP` code. In particular, we will represent the work tape by $\mathbb{N}$-variables, and we will update the configuration of the work tape by pure `LOOP` code. This is possible since the language is powerful enough to compute any primitive recursive function. In particular, `LOOP` programs can perform arithmetical operations like multiplication, addition, subtraction and integer division. Let b denote the cardinality of the set $\Gamma \cup \{B\}$, that is, the number of different symbols M can write to its work tape. Let $\alpha : \{0,\dots,b-1\} \to \Gamma \cup \{B\}$ be a bijection, and let the number $i < b$ represent the symbol $\alpha(i)$. We represent the stack $\alpha(x_1),\dots,\alpha(x_n)$, where $\alpha(x_1)$ is the top element and $\alpha(x_n)$ is the bottom element, by the natural number

$$x_1 + x_2b^1 + x_3b^2 + \dots + x_nb^{n-1},$$

i.e. as an n-digit number in base b. We implement the stack operations by their obvious arithmetical correspondents, e.g. the pop operation corresponds to integer division by b. (We need a status variable to distinguish the empty stack from the stack only containing the single element 0.) Hence, we conclude that the body of the `while` loop and the initiation part `init` can be implemented in a mixture of `LOOP` and `CLIP` code.

Let `body` denote the implementation of the loop body, and let `init'` denote the implementation of `init`. Note that since M works in logarithmic space and does terminate, there exist fixed $k_1, m_1 \in \mathbb{N}$ such that the number of times the body of the `while` loop in $\mathtt{P}_0$ will be executed is bounded by $\max(|x|, k_1)^{m_1}$ where $|x|$ denotes the length of the input. Let $\mathtt{W}_1, \dots, \mathtt{W}_{m_1}$ be fresh $\mathbb{W}$-variables, and let $\mathtt{Z}_1, \dots, \mathtt{Z}_{m_1}$ be fresh $\mathbb{N}$-variables. Furthermore, let `setoutput` be code setting the $\mathbb{W}$-variable `Y` to the empty word ε if M is in the state q_A, and to the word `0` if M is in the state q_R. Finally, let $\mathtt{P}_1 \equiv$

```
init'; Z_1:= k_1; ... Z_{m_1}:= k_1; W_1:= X; ... W_{m_1}:= X;
loop Z_1[...loop Z_{m_1}[foreach W_1 [... foreach W_{m_1} [body] ...]] ...];
setoutput
```

Then $\{\mathtt{X} = x\}\mathtt{P}_1\{\mathtt{Y} = \varepsilon\} \Leftrightarrow M$ accepts x.

(4.3) Let a be the largest number held by an $\mathbb{N}$-variable during the execution of $\mathtt{P}_1$ on input $\mathtt{X} = x$. We have $a < \max(|x|, k_2)^{m_2}$ for some fixed $k_2, m_2 \in \mathbb{N}$.

We prove (4.3). The work tape is simulated by stacks. These stacks are represented by natural numbers held by $\mathbb{N}$-variables. It is easy to see that a in (4.3) represents a stack. Now, a stack of height n is represented as an n-digit number in base b (where b is the cardinality of the set $\Gamma \cup \{B\}$). Since M works in space $k_0 \log_2(|x| + 2)$, we have $a < b^{k_0 \log_2(|x|+2)+1}$. Now, b and k_0 are fixed numbers, i.e. they do not depend on the input x, and thus (4.3) follows by standard number-theoretic reasoning.

Next we transform $\mathtt{P}_1$ into a program $\mathtt{P}_1'$ such that (i) for some fixed $k_3 \in \mathbb{N}$ no $\mathbb{N}$-variable in $\mathtt{P}_1'$ exceeds $\max(|x|, k_3)$ during an execution of $\mathtt{P}_1'$ on input $\mathtt{X} = x$ and (ii) $\{\mathtt{X} = x\}\mathtt{P}_1\{\mathtt{Y} = \varepsilon\}$ iff $\{\mathtt{X} = x\}\mathtt{P}_1'\{\mathtt{Y} = \varepsilon\}$. That such a transformation is possible follows from (4.3) and a straightforward generalization of Lemma 3.4. Let `A` and `M` be fresh $\mathbb{N}$-variables, and let $\mathtt{P}_2$ be $\mathtt{P}_1'$ where each subprogram of the form `suc(Z)` is replaced by the subprogram

```
A:= M; loop Z [pred(A)]; pred(A); Z:= M; loop A [pred(Z)].
```

Then we have

$$\{\mathtt{X}=x,\mathtt{M}=m\}\mathtt{P}_2\{\mathtt{Y}=\varepsilon\} \;\Leftrightarrow\; M \text{ accepts } x$$

whenever $m \geq \max(|x|,k_3)$. Besides, imperatives of the form `suc(Z)` do not occur in $\mathtt{P}_2$.

We will now transform $\mathtt{P}_2$ into a `CLIP` program. Let $\mathtt{P}_2'$ be $\mathtt{P}_2$ where each subprogram of the form `pred(Z)` is replaced by `clip(Z)`; each occurrence of a subprogram of the form `Z:= c`, where `c` denotes $c \in \mathbb{N}$, is replaced by `Z:= w` where `w` denotes a word of length c; each occurrence of the syntactical element `loop` is replaced by the syntactical element `foreach`. Now, $\mathtt{P}_2'$ is a `CLIP` program and

$$\{\mathtt{X}=x,\mathtt{M}=m\}\mathtt{P}_2'\{\mathtt{Y}=\varepsilon\} \;\Leftrightarrow\; M \text{ accepts } x$$

whenever $|m| \geq \max(|x|,k)$. Further, there exists a `CLIP` program `lengthmax` such that

$$\{\mathtt{X}=x\}\mathtt{lengthmax}\{\mathtt{M}=m \text{ where } |m| = \max(|x|,k)\}.$$

(To construct `lengthmax` use the idea in the proof of Lemma 3.3.) Finally, let $\mathtt{P} \equiv \mathtt{lengthmax;}\,\mathtt{P}_2'$. Then `P` is a `CLIP` program and

$$\{\mathtt{X}=x\}\mathtt{P}\{\mathtt{Y}=\varepsilon\} \;\Leftrightarrow\; M \text{ accepts } x.$$

This completes the proof. □

LEMMA 4.4. *Every function computed by a `CLIP` program is in the algebra* $[I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$.

PROOF. Let `P` be a `CLIP` program where $\mathcal{V}(\mathtt{P}) = \vec{\mathtt{X}}$. We will prove by induction on the build-up of `P` that there exist $\vec{f} \in [I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$ such that $\{\vec{\mathtt{X}}=\vec{x}\}\mathtt{P}\{\vec{\mathtt{X}}=\vec{f}(\vec{x})\}$.

The cases $\mathtt{P} \equiv \mathtt{X:=\ Y}$, $\mathtt{P} \equiv \mathtt{X:=\ k}$ and $\mathtt{P} \equiv \mathtt{Q;R}$ are similar to the corresponding cases in the proof of Lemma 3.5.

Assume $\mathtt{P} \equiv \mathtt{clip(X)}$. Let $f(\varepsilon) = C_\varepsilon$, $f(0x) = I_1^2(x, f(x))$ and $f(1x) = I_1^2(x, f(x))$. Then we have $\{\mathtt{X}=x\}\mathtt{P}\{\mathtt{X}=f(x)\}$ and $f \in [I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$.

Assume $\mathtt{P} \equiv \mathtt{if\ left(X)=0\ [Q]}$ where $\mathtt{X} \notin \mathcal{V}(\mathtt{Q})$. The induction hypothesis yields functions $\vec{g} \in [I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$ such that $\{\vec{\mathtt{Y}}=\vec{y}\}\mathtt{Q}\{\vec{\mathtt{Y}}=\vec{g}(\vec{y})\}$. Let $c(z_1,z_2,\varepsilon) = I_2^2(z_1,z_2)$, $c(z_1,z_2,0x) = I_1^4(z_1,z_2,x,c(z_1,z_2,x))$ and $c(z_1,z_2,1x) = I_2^4(z_1,z_2,x,c(z_1,z_2,x))$. Then $c \in [I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$ and $c(z_1,z_2,x) = z_1$ if $x = 0y$ for some $y \in \mathbb{W}$, otherwise $c(z_1,z_2,x) = z_2$. Use the function c, the functions $\vec{g}$, projection functions and the composition scheme to generate functions $\vec{f} \in [I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$ such that $\{\vec{\mathtt{Z}}=\vec{z}\}\mathtt{P}\{\vec{\mathtt{Z}}=\vec{f}(\vec{z})\}$ where $\vec{\mathtt{Z}} = \vec{\mathtt{Y}},\mathtt{X}$.

Assume $\mathtt{P} \equiv$ `if left(X)=1 [Q]` where $\mathtt{X} \notin \mathcal{V}(\mathtt{Q})$. This case is analogous to the preceding case.

Assume $\mathtt{P} \equiv$ `foreach X [Q]` where $\mathtt{X} \notin \mathcal{V}(\mathtt{Q})$. Generalize the case $\mathtt{P} \equiv$ `loop X [Q]` in the proof of Lemma 3.5 to recursion on notation. This can be done straightforwardly. □

LEMMA 4.5. *Let* $f \in [I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$. *There exists a fixed* $m \in \mathbb{N}$ *such that if* $f(v_1, \dots, v_n) = w$ *and* $|w| > m$, *then* w *equals an end segment of one of the words* $v_1, \dots, v_n$.

PROOF. Let $C'_{\mathbb{W}}$ be a finite subset of $C_{\mathbb{W}}$ such that $f \in [I, C'_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$. Let E be the least set such that (1) $C'_{\mathbb{W}} \cup \{v_1, \dots, v_n\} \subseteq E$ and (2) if $x \in E$ and y is an end segment of x, then $y \in E$. We prove

$$\vec{x} \in E \;\Rightarrow\; f(\vec{x}) \in E \tag{4.6}$$

by induction on the build-up of f from the functions in $[I, C'_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$. If $f \in I \cup C'_{\mathbb{W}}$, then (4.6) holds trivially. If f is generated by composition (COMP), then (4.6) follows straightforwardly from the induction hypothesis. If f is generated by simultaneous recursion (SIMN) a second induction is required: Let us say f is f_1 where

$$\begin{aligned} f_i(\vec{x}, \varepsilon) &= g_i(\vec{x}),\\ f_i(\vec{x}, 0y) &= h_i^0(\vec{x}, y, f_1(\vec{x}, y), \dots, f_k(\vec{x}, y)),\\ f_i(\vec{x}, 1y) &= h_i^1(\vec{x}, y, f_1(\vec{x}, y), \dots, f_k(\vec{x}, y)), \end{aligned}$$

for $i = 1, \dots, k$. We prove by induction on the build-up of y that $\vec{x}, y \in E \Rightarrow f_i(\vec{x}, y) \in E$ for $i = 1, \dots, k$. *Case* $y = \varepsilon$. Then $f_i(\vec{x}, y) = g_i(\vec{x})$. Assume $\vec{x}, y \in E$. Then we have $f_i(\vec{x}, y) \in E$ by the induction hypothesis on g_i. *Case* $y = 0z$. Then $f_i(\vec{x}, y) = h_i^0(\vec{x}, z, f_1(\vec{x}, z), \dots, f_k(\vec{x}, z))$. Now, assume $\vec{x}, y \in E$. Then we also have $z \in E$ since z is an end segment of y. By the induction hypothesis on z we have $f_1(\vec{x}, z), \dots, f_k(\vec{x}, z) \in E$. Finally, by the induction hypothesis on h_i^0 we have $f_i(\vec{x}, y) \in E$. *Case* $y = 1z$. This case is similar to the preceding one. This proves (4.6).

Let m be such that $m \geq |u|$ for every $u \in C'_{\mathbb{W}}$. Assume $f(v_1, \dots, v_n) = w$ and $|w| > m$. By (4.6) we have $w \in E$, and since $|w| > m$, the word w cannot be an end segment of one of the words in $C'_{\mathbb{W}}$. Hence, w is an end segment of one of the words $v_1, \dots, v_n$. □

THEOREM 4.7. *A language is in* LOGSPACE *if and only if it can be decided by a function in* $[I, C_{\mathbb{W}}; \text{COMP}, \text{SIMN}]$.

PROOF. That any language in LOGSPACE can be decided by a function in the algebra follows from Lemma 4.2 and Lemma 4.4.

To prove the equivalence the other way around, assume f is a unary function in the algebra. We will argue that the language decided by f, i.e. the language $\{x \mid f(x) = \varepsilon\}$, can be decided by a Turing machine working in logarithmic space.

By Lemma 4.5 we have a fixed $m \in \mathbb{N}$ such that if $|f(v)| > m$ then $f(v)$ equals an end segment of v. It is easy to construct an (informal) algorithm computing f using a fixed number of registers storing words, and if a register during the computation of $f(v)$ stores w where $|w| > m$, then w equals an end segment of v. The Turing machine will represent each register by a string of bits and blanks in the form $bc_1 \dots c_m d_1 \dots d_k$ where $k = \lceil \log_2(|v| + 1) \rceil + 1$. The symbol b is a status bit telling if the length of the word held by the register is less than or equal to m; if so, the string $c_1 \dots c_m$ stores the word (an end segment of $c_1 \dots c_m$ might be blanks); if not, the word held by the register is an end segment of the input, and the bits $d_1 \dots d_k$ give the length of the end segment in binary notation (and hence the Turing machine can read the word from the input tape). The Turing machine starts the computation of f on input v by marking off a fixed number of areas on its work tape, one area for each register, each area occupying $1 + m + \lceil \log_2(|v| + 1) \rceil + 1$ tape cells excluding the markers. Then the Turing machine follows the algorithm and computes the value $f(v)$. Finally, the Turing machine checks the value $f(v)$ which will be stored in a particular register; if the value is the empty string, the Turing machine passes on to the accept state q_A; otherwise it passes on to the reject state q_R. It is obvious that a Turing machine designed along these lines will work in logarithmic space, and there should be no need to carry out the construction in details. The Turing machine will need some space for bookkeeping purposes, but the number of tape cells required will be small compared to the number of tape cells required to represent the registers. □

COROLLARY 4.8. *A language A can be decided by a Turing machine working in logarithmic space if and and only if $A = \{x \mid f(x) = \varepsilon\}$ for some function f defined from constant functions ($C_{\mathbb{W}}$) and projection functions (I) by composition and simultaneous recursion on notation.*

5. Some references

There are some results in the literature comparable to our characterizations of LOGSPACE and LINSPACE. Clote (1990) characterizes the complexity class AC^0 by a function algebra that contains neither bounds nor variable segregation. Ishihara (1999) characterizes the polytime functions by a similar function algebra. The operators in Clote's algebra are composition and concatenation recursion, whereas Ishihara's operators are composition and a stronger version of concatenation recursion (called full concatenation recursion). In contrast to us, Clote and Ishihara use complicated initial functions including the smash function. Jones (1999) characterizes LOGSPACE by a programming language reminiscent of our language CLIP. A pure recursion-theoretic characterization of LOGSPACE, similar to the one given in Corollary 4.8 above, should be within range of the results in Jones (1999). However, Jones does no attempt to state such a characterization. Neither is his programming language tailored to yield one. (Our CLIP language is specially tailored for this purpose.)

The results in this paper originated from our studies of imperative programming languages. It seems fruitful to integrate complexity theory and programming language theory. See Jones (1997, 1999, 2001), Irwin *et al.* (2001), Kristiansen & Niggl (2004), and Kristiansen & Voda (2003).

Lind (1974) is the first author who characterizes LOGSPACE by a function algebra (containing explicit bounds). Bellantoni (1992) characterizes LOGSPACE by a function algebra where the definition schemes distinguish between safe and normal arguments. Bellantoni (1992) also characterizes LINSPACE by such a function algebra. Ritchie (1963) is the first author characterizing LINSPACE by a function algebra (containing explicit bounds). As mentioned above, Ritchie proves that the class of number-theoretic functions computable by a Turing machine working in linear space equals the Grzegorczyk class $\mathcal{E}^2$. See Clote (1999) for more references and the historical details.

Acknowledgements

The author wants to thank the referees and the editor for valuable advice regarding the presentation.

References

S. J. BELLANTONI (1992). Predicative recursion and computational complexity. Technical Report 264/92, University of Toronto, Computer Science Department.

S. J. Bellantoni & S. Cook (1992). A new recursion-theoretic characterization of the polytime functions. *Comput. Complexity* **2**, 97–110.

P. Clote (1990). Sequential, machine-independent characterizations of parallel complexity classes ALOGTIME, AC^k, NC^k and NC. In *Feasible Mathematics*, S. Buss and P. J. Scott (eds.), Birkhäuser, 49–70.

P. Clote (1999). Computation models and function algebras. In *Handbook of Computability Theory*, E. Griffor (ed.), Elsevier, 589–681.

A. Cobham (1965). The intrinsic computational difficulty of functions. In *Logic, Methodology and Philosophy of Science* (Proc. 1964 Internat. Congr.), North-Holland, 24–30.

D. Hilbert & P. Bernays (1934). *Grundlagen der Mathematik*. Springer, Berlin.

R. J. Irwin, B. M. Kapron & J. S. Royer (2001). On characterizations of the basic feasible functionals. I. *J. Funct. Programming* **11**, 117–153.

H. Ishihara (1999). Function algebraic characterization of the polytime functions. *Comput. Complexity* **8**, 346–356.

N. D. Jones (1997). *Computability and Complexity from a Programming Perspective*. MIT Press, Cambridge, MA.

N. D. Jones (1999). logspace and ptime characterized by programming languages. *Theoret. Comput. Sci.* **228**, 151–174.

N. D. Jones (2001). The expressive power of higher-order types or, life without CONS. *J. Funct. Programming* **11**, 55–94.

L. Kristiansen & K.-H. Niggl (2004). On the computational complexity of imperative programming languages. *Theoret. Comput. Sci.* **318**, 139–161.

L. Kristiansen & P. J. Voda (2003). Complexity classes and fragments of C. *Inform. Process. Lett.* **88**, 213–218.

D. Leivant (1991). A foundational delineation of computational feasibility. In *IEEE 6th Annual Symposium on Logic in Computer Science*, IEEE, 39–47.

D. Leivant (1993). Stratified functional programs and computational complexity. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM, New York, 325–333.

J. C. Lind (1974). Computing in logarithmic space. Technical report, Project MAC Technical Memorandum 52, Massachusetts Institute of Technology.

A. R. Meyer & D. M. Ritchie (1967). The complexity of loop programs. In *Proc. ACM Nat. Conf.*, 465–469.

P. Odifreddi (1999). *Classical Recursion Theory. Vol. II.* Studies in Logic Found. Math. 143 North-Holland, Amsterdam.

R. Péter (1957). *Rekursive Funktionen.* Verlag der Ungarischen Akademie der Wissenschaften, Budapest. English translation: Academic Press, New York, 1967.

R. W. Ritchie (1963). Classes of predictably computable functions. *Trans. Amer. Math. Soc.* **106**, 139–173.

H. Simmons (1988). The realm of primitive recursion. *Arch. Math.* **27**, 177–188.

Manuscript received September 2002

Lars Kristiansen
Oslo University College
Faculty of Engineering
Cort Adelers gate 30
N-0254 Oslo, Norway
`larskri@iu.hio.no`
`http://www.iu.hio.no/~larskri`

Department of Mathematics
University of Oslo
Postboks 1053, Blindern
N-0316 Oslo, Norway