Programming Languages Capturing Complexity Classes

Lars Kristiansen[1,2] and Paul J. Voda[3]

[1] Oslo University College, Faculty of Engineering
larskri@iu.hio.no WWW home page: http://www.iu.hio.no/~larskri
[2] Department of Mathematics, University of Oslo
[3] Institute of Informatics, Comenius University Bratislava
voda@fmph.uniba.sk WWW home page: http://www.fmph.uniba.sk/~voda

Abstract. We investigate an imperative and a functional programming language. The computational power of fragments of these languages induce two hierarchies of complexity classes. Our first main theorem says that these hierarchies match, level by level, a complexity-theoretic alternating space-time hierarchy known from the literature. Our second main theorems says that a slightly different complexity-theoretic hierarchy (the Goerdt-Seidl hierarchy) also can be captured by hierarchies induced by fragments of the programming languages.

1 Introduction

In this paper we relate the computational power of natural fragments of pure and simple programming languages to complexity classes defined by Turing machines and explicit resource bounds. The elements of the languages we investigate constitute an essential core of many real-life programming languages. We study a functional programming language which is simply the typed λ-calculus extended with recursors and numerals. The type λ-calculus with its syntax and semantics is an essential part of languages like Lisp, ML, Haskell, etc. Further, we study an imperative language with syntax and semantics reminiscent of e.g. Pascal and C. Higher types is an essential constituents in both languages.

We think the results presented in this paper are interesting for several reasons. *(1) Complexity theory and programming language theory:* We subscribe to the thesis that complexity and programming language theories have much to offer each other, in both directions (Jones [12] [13]). Our results somewhat bridge the gap between the two areas. Moreover, they also provide some evidence for the thesis by giving an analysis of the expressive power of elements of imperative languages versus the expressive power of elements of functional languages. This analysis is founded on complexity theory. *(2) The naturalness of the complexity classes:* During the last three or four decades researchers have provided so called resource-free and intrinsic characterizations of the complexity classes, e.g. the seminal papers of Cobham [8], Fagin [9] and Bellantoni-Cook [5]. One of the motivations is to show that the complexity classes, defined by a particular machine model and explicit resource bounds, are natural entities, and that complexity theory is a robust and mathematically interesting field. Our theorems provide yet another argument in favor of the naturalness of the complexity classes. *(3) Open problems in complexity theory:* The characterizations of the complexity classes given paper might shed some light upon some of the many open questions regarding the relationship between these classes. *(4) Goerdt and Seidl's work:* Some work of Goerdt and Seidl's is less known than it deserves to be. We remedy the situation by paying attention to the results and techniques from [11] and [10]. *(5) Pedagogical value:* Many scientists and researchers familiar with programming languages and programming theory, are not familiar with complexity theory. This paper might be informative and enlightening to such an audience and e.g. help them to acquire a better understanding of open problems in complexity theory.

This paper is organized as follows. In Section 2 we define the programming languages and state the first of two main theorems. The three subsequent sections give the proof this theorem. These sections are occasionally very technical, and some readers might want to skip them and go straight ahead to Section 6 where we state the second main theorem and make a summary. (The proofs of the two theorem are uniform, and hence, we will not give a proof of the second theorem.) In the final section we discuss some related research.

The results in this paper are generalization and refinements of the results published in Kristiansen-Voda [18] and Kristiansen-Voda [17]. The authors want to thank G. Mathias Barra for useful comments and the University of Copenhagen for hospitality.

2 The Programming Languages and the Main result

2.1 Turing Machines and the Alternating Space-Time Hierarchy

We will assume that the read is familiar with Turing machines and basic complexity theory. For more on the subject see e.g. Odifreddi [23] or Lewis-Papadimitriou [21].

Definition. A Turing machine M *decides* a problem A when M on input $x \in \mathbb{N}$ halts in a distinguished accept state if $x \in A$, and in a distinguished reject state if $x \notin A$. The input $x \in \mathbb{N}$ should be represented binary on the Turing machine's input tape. We will use $|x|$ to denote the length of the (shortest possible) binary representation of the natural number x. For $i \in \mathbb{N}$, we define $\text{TIME } 2_i^{\text{LIN}}$ ($\text{SPACE } 2_i^{\text{LIN}}$) to be the set of problem decidable by a deterministic Turing machine working in time (space) $2_i^{c|x|}$ for some fixed $c \in \mathbb{N}$. (The notation 2_i^x is as usual defined by $2_0^x = x$ and $2_{i+1}^x = 2^{2_i^x}$) □

It is trivial that $\text{TIME } 2_i^{\text{LIN}} \subseteq \text{SPACE } 2_i^{\text{LIN}}$ and $\text{SPACE } 2_i^{\text{LIN}} \subseteq \text{TIME } 2_{i+1}^{\text{LIN}}$, and thus, we have an *alternating space-time hierarchy*

$$\text{SPACE } 2_0^{\text{LIN}} \subseteq \text{TIME } 2_1^{\text{LIN}} \subseteq \text{SPACE } 2_1^{\text{LIN}} \subseteq \text{TIME } 2_2^{\text{LIN}} \subseteq \text{SPACE } 2_2^{\text{LIN}} \subseteq \text{TIME } 2_3^{\text{LIN}} \subseteq \dots .$$

The three classes at the bottom of the hierarchy are often called respectively LINSPACE, EXP, and EXPSPACE in the literature. It is well known, and quite obvious, that SPACE $2_i^{\text{LIN}} \subset$ SPACE 2_{i+1}^{LIN} and TIME $2_i^{\text{LIN}} \subset$ TIME 2_{i+1}^{LIN} hold for any $i \in \mathbb{N}$. Thus, at least one of the two inclusions

$$\text{SPACE } 2_i^{\text{LIN}} \subseteq \text{TIME } 2_i^{\text{LIN}} \subseteq \text{SPACE } 2_{i+1}^{\text{LIN}}$$

must be strict, but it is not known which one(s). Further, at least one of the two inclusions

$$\text{TIME } 2_i^{\text{LIN}} \subseteq \text{SPACE } 2_i^{\text{LIN}} \subseteq \text{TIME } 2_{i+1}^{\text{LIN}}$$

has to be strict, but it is not known which one(s). In particular, no one knows if LINSPACE is strictly included in EXP, and no one knows if EXP is strictly included in EXPSPACE. These notorious open problems are again closely related to the even more notorious problems of complexity theory, e.g. is LOGSPACE strictly included in P; is P strictly included in PSPACE? Complexity theorists in general, and the authors in particular, suspect all these inclusions to be strict, and it is a bit of a mystery that it should be so hard to find proofs. A further study of the two alternating space-time hierarchies might shed some light on these and similar pivotal questions of complexity theory (e.g. if we could prove for *some* $n \in \mathbb{N}$ that SPACE $2_n^{\text{LIN}} \neq$ TIME 2_{n+1}^{LIN} it follows that P $\neq$ LOGSPACE and EXP $\neq$ LINSPACE).

2.2 Numbers and Types

Definition. We will use small Greek letters, with or without decorations, to denote *types*. The *types* are defined recursively:

- $\mathbf{0}$ is a type
- $\sigma \to \tau$ is a type if σ and τ are types
- $\sigma \times \tau$ is a type if σ and τ are types.

We use $\sigma, \sigma' \to \sigma''$ as alternative notation for $\sigma \to (\sigma' \to \sigma'')$. We interpret $\sigma \to \sigma' \to \sigma''$ by associating parentheses to the right, i.e. as $\sigma \to (\sigma' \to \sigma'')$. We say a type σ is of *level* n when $\text{lv}(\sigma) = n$ where

- $\text{lv}(\mathbf{0}) = 0$
- $\text{lv}(\sigma \to \tau) = \max(\text{lv}(\sigma) + 1, \text{lv}(\tau))$
- $\text{lv}(\sigma \times \tau) = \max(\text{lv}(\sigma), \text{lv}(\tau))$.

We define the *cardinality of type* σ *at base* b, written $|\sigma|_b$, by recursion on the structure of σ: (i) $|\mathbf{0}|_b = b$, (ii) $|\rho \to \tau|_b = |\tau|_b^{|\rho|_b}$, and (iii) $|\rho \times \tau|_b = |\rho|_b \times |\tau|_b$. □

Lemma 1. *For any* $m, k \in \mathbb{N}$ *there exists a type* σ *of level* m *such that* $2_{m+1}^{k|x|} < |\sigma|_x$ *for all but finitely many values of* x.

Proof. First we prove by induction on m that for every polynomial p there exists a type σ of level m such that $2_m^{p(x)} < |\sigma|_x$ for all but finitely many values of x. *Case* $m = 0$. Let $\mathbf{0}^0 = \mathbf{0}$ and $\mathbf{0}^{j+1} = \mathbf{0} \times \mathbf{0}^j$. Then $p(x) < |\mathbf{0}^j|_x$ holds for all but finitely many values of x when j is sufficiently large. *Step:* Assume $|\sigma|_x > 2_m^{p(x)}$ where $\mathrm{lv}(\sigma) = m$. Then, we have $2_{m+1}^{p(x)} = 2^{(2_m^{p(x)})} < 2^{|\sigma|_x} < (|\mathbf{0}|_x)^{|\sigma|_x} = |\sigma \to \mathbf{0}|_x$ for all but finitely many values of x. Further, $\mathrm{lv}(\sigma \to \mathbf{0}) = \mathrm{lv}(\sigma) + 1$. The lemma follows since for every $k \in \mathbb{N}$ there exists polynomial p such that $2^{k|x|} < p(x)$. □

Lemma 2. *For every type σ of level k there exists a polynomial p such that $2_k^{p(x)} > |\sigma|_x$, and hence, there also exists $c \in \mathbb{N}$ such that $2_{k+1}^{c|x|} > |\sigma|_x$.*

Proof. We use induction on the structure of σ. The case $\sigma = \mathbf{0}$ is trivial. Assume $\sigma = \rho \to \tau$. Then we have $\mathrm{lv}(\rho) \leq n - 1$ and $\mathrm{lv}(\tau) \leq n$, and the induction hypothesis yields polynomials q and r such that $2_{n-1}^{q(x)} > |\rho|_x$ and $2_n^{r(x)} > |\tau|_x$. We have $|\sigma|_x = |\tau|_x^{|\rho|_x} < (2_n^{r(x)})^{2_{n-1}^{q(x)}} = 2^{(2_{n-1}^{r(x)} \times 2_{n-1}^{q(x)})} \leq 2_n^{r(x)+q(x)}$ and the lemma holds when $p = r + q$. (For any polynomial $p(x)$ there exists $c \in \mathbb{N}$ such that $2^{c|x|} > p(x)$.) □

Definition. The natural number a is a *number of type σ at base b*, in symbols $a : \sigma_b$, iff $a < |\sigma|_b$. Let $a : (\sigma \to \tau)_b$. Then a can be viewed as a $|\sigma|_b$ digit number in base $|\tau|_b$, and thus, a can be uniquely written in the form

$$v_0 \;+\; v_1|\tau|_b^1 \;+ \cdots +\; v_k|\tau|_b^k$$

where $k = |\sigma|_b - 1$ and $v_j : \tau_b$ for $j \in \{0, \ldots, k\}$. We call $v_0, \ldots, v_k$ the *digits* in a, and for any $i : \sigma_b$, we denote the i'th digit in a by $a[i]_b$, i.e. $a[i]_b = v_i$. Furthermore, for any $i : \sigma_b$ and $w : \tau_b$, let $a[i := w]_b$ denote the number which is the result of setting the i'th digit in a to w. (Note that $a[i := w]_b$ is a number of type σ at base b.) The notation $a[i_1, \ldots, i_n]_b$, where $n \geq 1$, abbreviates $((a[i_1]_b)[i_2]_b) \ldots)[i_n]_b$. Further, let

$$a[i_1, \ldots, i_{n+1} := w]_b \;\stackrel{\text{def}}{=}\; a[i_1, \ldots, i_n := a[i_1, \ldots, i_n]_b[i_{n+1} := w]_b]_b$$

for $n \geq 1$. Thus, $a[i_1, \ldots, i_n := w]_b$ is the number which is the results of setting the sub-digit $a[i_1, \ldots, i_n]_b$ in a to w. □

2.3 The Imperative Programming Language and the $\mathcal{I}$-Hierarchy

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 `P` is executed, then the condition given by the sentence B is fulfilled after the execution of `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 `X` and `Y`, respectively, before the execution of `P`, then the values x' and y' are held by `X` and `Y` after the execution of* `P`. We use typewriter style uppercase and lowercase letters, with or without subscripts, to denote respectively program variables and terms. Occasionally we indicate the type of a variable or a term by a superscript, e.g. $\mathtt{t}^\sigma$ signifies that the term $\mathtt{t}$ is of type σ. Another typical example is $\{\mathtt{X}_1 = x_1, \mathtt{X}_2 = x_2\}\,\mathtt{P}\,\{\mathtt{Y} = f(x_1, x_2)\}$ meaning that if the values x_1 and x_2 held by $\mathtt{X}_1$ and $\mathtt{X}_2$, respectively, before the execution of $\mathtt{P}$, then the value held by $\mathtt{Y}$ after the execution of $\mathtt{P}$ equals $f(x_1, x_2)$. 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.

We will call a type on the form $\tau \to \rho$ for an *array* type. The reason for introducing this terminology is obvious. Program variables of type $\tau \to \rho$ are similar to the arrays we know from programming languages like Pascal and C. A variable $\mathtt{X}$ of array type $\tau \to \rho$ has the entries $\mathtt{X[0]}$, $\mathtt{X[1]}$,..., $\mathtt{X[}k\mathtt{]}$ where $k = |\tau|_b$ for some b which remains fixed through the execution of the program. (The value of b is determined from the input when the program starts. Thus, the arrays are allocated dynamically at program start.) Each entry of the array holds a value in the set $\{0, 1, \ldots, |\rho|_b\}$. When a variable $\mathtt{X}$ of array type $\tau \to \rho$ holds a number $a : (\tau \to \rho)_b$, then the i'th entry of $\mathtt{X}$ will hold the i'th digit of a (when we view a as a $|\tau|_b$ digit number in base $|\rho|_b$).

Definition. First we define the syntax of the imperative programming language.

The type σ is an *array type* if $\sigma \equiv \tau \to \rho$ for some types τ and ρ. The language has an infinite supply of *program variables* for each *array type* σ. Further, the language has an infinite supply of *program variables* of type $\mathbf{0}$, in particular, there is a type $\mathbf{0}$ variable $\mathtt{x}^{\mathbf{0}}_{\mathrm{input}}$ dedicated to hold the input. Any variable of type σ is a *term* of type σ; $\mathtt{t[X]}$ is an is an term of type τ if $\mathtt{X}$ is a variable of type σ and $\mathtt{t}$ is a term of type $\sigma \to \tau$. We use $\mathtt{X[Y}_1, \ldots, \mathtt{Y}_n\mathtt{]}$ to abbreviate $\mathtt{X[Y_1][Y_2]}\ldots\mathtt{[Y}_n\mathtt{]}$. (Thus a term is a single variable, or it has the form $\mathtt{X[Y}_1, \ldots, \mathtt{Y}_n\mathtt{]}$ where $\mathtt{X}, \mathtt{Y}_1, \ldots, \mathtt{Y}_n$ are variables.) The syntax of a program is given by

$$
\begin{array}{lcl}
\mathtt{t} \in \text{Term} & ::= & \text{term of type } \mathbf{0} \\
\mathtt{s} \in \text{Statement} & ::= & \mathtt{accept} \mid \mathtt{t+} \mid \mathtt{if\,t\,\{p\}} \mid \mathtt{for}_\sigma\,\mathtt{\{p\}} \ \ \text{(for any type } \sigma) \\
\mathtt{p} \in \text{Program} & ::= & \mathtt{s} \mid \mathtt{s;p}
\end{array}
$$

Program on the form $\mathtt{for}_\sigma\,\mathtt{\{p\}}$ will occasionally be called a *loops.*

We will now give the programming language's semantics.

The input to a program is a single natural number x, and the variable $\mathtt{x}^{\mathbf{0}}_{\mathrm{input}}$ holds x when the execution starts. A program is executed in a particular base b. When the execution starts, the base is set to $b = \max(x, 1) + 1$ where x is the program's input. The base will not be modified during the execution. Program variables of type σ store natural numbers in the set $\{0, 1, \ldots, |\sigma|_b - 1\}$. The primitive instruction $\mathtt{accept}$ does not modify any variables, i.e. we have $\{\mathtt{X} = x\}\,\mathtt{accept}\,\{\mathtt{X} = x\}$ for any variable $\mathtt{X}$. The only primitive instruction which is capable of modifying the variables, has either the form (i) $\mathtt{X+}$ where $\mathtt{X}$ is a type $\mathbf{0}$ variable, or (ii) the form $\mathtt{X[Y}_1, \ldots, \mathtt{Y}_n\mathtt{]+}$ where $\mathtt{X}, \mathtt{Y}_1, \ldots, \mathtt{Y}_n$ are variables such that $\mathtt{X[Y}_1, \ldots, \mathtt{Y}_n\mathtt{]}$ is a type $\mathbf{0}$ term. (Any other form is rule out by the syntax rules.) In case (i) $\mathtt{X}$ is increased by 1 modulo b, i.e. we have

$$\{\mathtt{X} = x\}\,\mathtt{X+}\,\{\mathtt{X} = x + 1 \pmod{b}\}$$

and $\{\mathtt{Z} = z\}\,\mathtt{X+}\,\{\mathtt{Z} = z\}$ for every variable Z different from X. In case (ii) the array X is modified such that the entry $\mathtt{X}[\mathtt{Y}_1, \ldots, \mathtt{Y}_n]$ is increased by 1 modulo b, i.e. let $v = x[y_1, \ldots, y_n] + 1 \pmod b$, then

$$\{\mathtt{X} = x, \mathtt{Y}_1 = y_1, \ldots, \mathtt{Y}_n = y_n\}\,\mathtt{X}[\mathtt{Y}_1, \ldots, \mathtt{Y}_n]\mathtt{+}\,\{\mathtt{X} = x[y_1, \ldots, y_n := v]\}$$

and $\{\mathtt{Z} = z\}\,\mathtt{X}[\mathtt{Y}_1, \ldots, \mathtt{Y}_n]\mathtt{+}\,\{\mathtt{Z} = z\}$ for every variable Z different from X. The program $\mathtt{for}_\sigma\,\{\,\mathtt{p}\,\}$ executes the program p; p; ...p where p is repeated $|\sigma|_b$ times. The program if t { p } executes the program p if the value held by t is different from 0; otherwise the program does nothing. The semicolon composing two programs has the standard meaning. □

The semantics given above is more than precise enough for our purposes, namely to define the class $\mathcal{I}^{i,j}$ for $i, j \in \mathbb{N}$. It follows strait forwardly from the definition that $\mathcal{I}^{i,j} \subseteq \mathcal{I}^{i+1,j}$ and $\mathcal{I}^{i,j} \subseteq \mathcal{I}^{i,j+1}$ for any $i, j \in \mathbb{N}$. Thus we have a hierarchy

$$\mathcal{I}^{0,0} \subseteq \mathcal{I}^{0,1} \subseteq \mathcal{I}^{1,1} \subseteq \mathcal{I}^{1,2} \subseteq \mathcal{I}^{2,2} \subseteq \mathcal{I}^{2,3} \subseteq \ldots$$

Definition. A program *accepts* its input $x \in \mathbb{N}$ if it executes the primitive statement accept ; otherwise the program *rejects* its input. A program *decides* the problem A when the program accepts the input x if $x \in A$; and rejects the input x if $x \notin A$. A *term* is of *rank* n when the type of every variable occurring in the term is of level $\leq n$. A *program* is of *data rank* n when every term t occurring in a subprogram on the form t+ is of rank $\leq n$. A loop $\mathtt{for}_\sigma\,\{\,\mathtt{p}\,\}$ is of *rank* n when $n \leq \mathrm{lv}(\sigma)$. A program is of *loop rank* n when every loop occurring in the program is of rank n. A problem A belongs to the class $\mathcal{I}^{i,j}$ iff A is decided by some program of loop rank i and data rank j. □

2.4 The Functional Programming Language and the Hierarchy $\mathcal{F}$

Our functional programming language will be an extension the standard typed λ-calculus.

Definition. We define the *terms* of *the standard typed* λ*-calculus.*

- We have an infinite supply of variables $x_0^\sigma, x_1^\sigma, x_2^\sigma, \ldots$ for each type σ. A variable of type σ is a term of type σ;
- $\lambda x M$ is a term of type $\sigma \to \tau$ if σ is a variable of type σ and M is a term of type τ *(λ-abstraction)*
- (MN) is a term of type τ if M is a term of type $\sigma \to \tau$ and N is a term of type σ *(application)*
- $\langle M, N\rangle$ is a term of type $\sigma \times \tau$ if M is a term of type σ and N is a term of type τ *(product)*
- $\mathbf{fst}M$ $(\mathbf{snd}M)$ is a term of type σ (τ) if M is a term of type $\sigma \times \tau$ *(projections).*

The reduction rules of the standard typed λ-calculus are the usual ones. We have $(\lambda x M)N \rhd M[x := N]$ if $x \notin FV(N)$ (β-conversion); $\mathbf{fst}\langle M, N\rangle \rhd M$; and $\mathbf{snd}\langle M, N\rangle \rhd N$; $\lambda x(Mx) \rhd M$ if $x \notin FV(M)$; $\langle \mathbf{fst}M, \mathbf{snd}M\rangle \rhd M$; We will call first three rules for *β-conversions*, and the two last rules for *η-conversions*. Further, we have the usual reduction rules $(MN) \rhd (MN')$ if $N \rhd N'$; $(M'N) \rhd (M'N)$ if $M \rhd M'$; ...etcetera. When it is not possible to use β-conversions or η-conversions in a subterm of M, we will say that M is in the *$\beta\eta$-normal form*.

The calculus T^- is the standard typed λ-calculus extended with the constant $1 : \mathbf{0}$, and for each type σ the *recursor* R_σ of type $\sigma, \mathbf{0} \to \sigma \to \sigma, \mathbf{0} \to \sigma$.

The calculus T is the calculus T^- extended with the constants $0 : \mathbf{0}$ (zero) and $s : \mathbf{0} \to \mathbf{0}$ (successor), the reduction rule $1 \rhd s0$, and for each type σ, the reduction rules $R_\sigma(P, Q, 0) \rhd P$ and $R_\sigma(P, Q, sN) \rhd Q(N, R_\sigma(P, Q, N))$. We use $\overline{n}$ to denote the *numeral* $s^n 0$ where $s^0 0 = 0$ and $s^{n+1} 0 = (ss^n 0)$.

We will use $=$ to denote the symmetric-transitive-reflexive closure of $\rhd$. We will say that M and N are *equal* when $M = N$. □

We will assume that the reader is familiar with the typed λ-calculus, and we will use the standard conventions in the literature, e.g. $F(X, Y)$ means $((FX)Y)$; $\lambda xyz.M$ means $\lambda x \lambda y \lambda z M$. Occasionally, we omit parenthesizes, and a term like $MNPQ$ should be read the way that make sense according to the typing. Occasionally, we will indicate the types of terms and variables by superscripts, e.g. $\lambda x^{\mathbf{0}} y^\sigma . M^\tau$ indicates that the variables x and y have, respectively, type $\mathbf{0}$ and τ, and that the term M has type τ. We will assume that variables are renamed in reductions such that we avoid trivial name conflicts. For more on the λ-calculus see e.g. Sørensen-Urzyczyn [30] or Simmons [29].

Note that the successor s cannot occur in a T^--term. Further, note that the calculus T^- has no reduction rules in addition to those of the standard typed λ-calculus, and that e.g. the term $\mathrm{R}_\sigma(M, N, 1)$ is irreducible in the calculus T^- if M and N are irreducible.

It is well known that any closed T-term of type $\mathbf{0}$ normalizes to a unique numeral. Thus, a closed term M of type $\mathbf{0} \to \mathbf{0}$ defines a function $f : \mathbb{N} \to \mathbb{N}$, and the value $f(n)$ can be computed by normalizing the term $M\overline{n}$. Any function provably total in Peano Arithmetic is definable in T. (See [1] for more on the T-calculus and Gödel's T.) If we disallow occurrences of the successor s in the defining terms, the class of functions definable is of course severely restricted. Indeed, at a first glance it is hard to believe that any interesting functions at all can be defined without the successor function.

Definition. A *functional program* is a closed T^--term of type $\mathbf{0} \to \mathbf{0}$. The functional program $M : \mathbf{0} \to \mathbf{0}$ *decides* the problem A when $M\overline{n} = \overline{0} \Leftrightarrow n \in A$. The *recursor rank* $\mathrm{rk}(M)$ of the T-term M equals the least $n \in \mathbb{N}$ such that for any recursor R_σ occurring in M we have $\mathrm{lv}(\sigma) \leq n$. (Note that any functional program is a T-term, and thus has a recursor rank.) A problem A belongs to the class $\mathcal{F}^i$ iff A can be decided by a functional program of recursor rank $\leq i$. □

It follows straightforwardly from the definitions that we have a hierarchy $\mathcal{F}^0 \subseteq \mathcal{F}^1 \subseteq \mathcal{F}^2 \subseteq \ldots$.

2.5 The First Main Theorem

We are ready to state the first main theorem.

Theorem 1. *We have* SPACE $2_i^{\text{LIN}} = \mathcal{I}^{i,i} = \mathcal{F}^{2i}$ *and* TIME $2_{i+1}^{\text{LIN}} = \mathcal{I}^{i,i+1} = \mathcal{F}^{2i+1}$ *for any* $i \in \mathbb{N}$.

We will spend the next three sections proving the theorem. In Section 3 we prove the inclusions SPACE $2_i^{\text{LIN}} \subseteq \mathcal{I}^{i,i}$ and TIME $2_{i+1}^{\text{LIN}} \subseteq \mathcal{I}^{i,i+1}$; in Section 4 we prove the inclusions SPACE $2_i^{\text{LIN}} \subseteq \mathcal{F}^{2i}$ and TIME $2_{i+1}^{\text{LIN}} \subseteq \mathcal{F}^{2i+1}$; and in Section 5 we prove the inclusions $\mathcal{F}^{2i} \subseteq$ SPACE 2_i^{LIN} and $\mathcal{F}^{2i+1} \subseteq$ TIME 2_{i+1}^{LIN}.

3 The Proof of SPACE $\mathbf{2_i^{\text{LIN}}} \subseteq \boldsymbol{\mathcal{I}^{i,i}}$ and TIME $\mathbf{2_{i+1}^{\text{LIN}}} \subseteq \boldsymbol{\mathcal{I}^{i,i+1}}$

Lemma 3 (Basic programs). *Let $b \geq 2$ denote the base of the execution. For any terms $\mathtt{t}^0$, $\mathtt{s}^0$, $\mathtt{u}^0$ we have the programs*

1. $\{\mathtt{t} = x\}\ \mathtt{t:=}\ell\ \{\mathtt{t} = \ell\}$
   for any $\ell < b$ (assignment of constants strictly less than the base)
2. $\{\mathtt{t} = x, \mathtt{s} = y\}\ \mathtt{s:=t}\ \{\mathtt{t} = x, \mathtt{s} = x\}$
   (assignment)
3. $\{\mathtt{t} = x, \mathtt{s} = y\}\ \mathtt{u:=t+s}\ \{\mathtt{u} = x + y \pmod b\}$
   (addition modulo the base)
4. $\{\mathtt{t} = x, \mathtt{s} = y\}\ \mathtt{u:=t-s}\ \{\mathtt{u} = x - y \pmod b\}$
   (subtraction modulo the base)
5. $\{\mathtt{t} = x\}\ \mathtt{s:=sg\,t}\ \{\mathtt{s} = x'\}$
   where $x' = 0$ if $x = 0$; otherwise $x' = 1$ (converting numbers to boolean values)
6. $\{\mathtt{t} = x\}\mathtt{s:=NOT\,t}\ \{\mathtt{s} = x'\}$
   where $x' = 1$ if $x = 0$; otherwise $x' = 0$ (logical **not***)*
7. $\{\mathtt{t} = x, \mathtt{s} = y\}\ \mathtt{u:=t\,OR\,s}\ \{\mathtt{u} = z\}$
   where $z = 0$ if $x = y = 0$; otherwise $z = 1$ (logical **or***)*
8. $\{\mathtt{t} = x, \mathtt{s} = y\}\ \mathtt{u:=t\,AND\,s}\ \{\mathtt{u} = z\}$
   where $z = 1$ if $x = y = 1$; otherwise $z = 0$ (logical **and***)*

Moreover, all the programs are of loop rank 0.

Proof. Let `while t { p }` denote the program $\mathtt{for}_0$ `{ if t { p } }`, so roughly speaking, the program `while t { p }` execute the loop's body `p` while $\mathtt{t} \neq 0$. Let $\mathtt{t:=0} \equiv$ `while t {t+}` and let $\mathtt{t:=}\ell + 1 \equiv \mathtt{t:=}\ell\mathtt{;\,t+}$. Let $\overline{x}$ denote the complement of x modulo b, i.e. $\overline{x}$ is the unique number such that $x + \overline{x} = 0 \pmod b$. Let

$$\mathtt{cc(t,s,u)} \;\equiv\; \mathtt{s:=0;\ u=0;\ while\ t\ \{\,t+;\ s+;\ u+\,\}}\,.$$

Then we have

$$\{\mathtt{t} = x, \mathtt{s} = y, \mathtt{u} = z\}\,\mathtt{cc(t,s,u)}\{\mathtt{t} = 0, \mathtt{s} = \overline{x}, \mathtt{u} = \overline{x}\}$$

Note that $\overline{\overline{x}} = x$. Thus, let $\mathtt{s:=t} \equiv \mathtt{cc(t,X,Y);cc(X,t,s)}$ where `X` and `Y` are fresh variables of type **0**. Further, let

$$\mathtt{u:=t+s} \;\equiv\; \mathtt{u:=t;\ cc(s,X,Y);\ while\ X\ \{\,u+;\ X+\,\}}$$

where X and Y are fresh variables of type $\mathbf{0}$. By using the program we have defined so far we can easily define the program `u:=t-s` since $x-y = \overline{\overline{x}+y} \pmod b$. Let `s:=sg t` $\equiv$ `s:=0; if t { s:=1 }` and `s:=NOT t` $\equiv$ `s:=1; if t { s:=0 }`. Let s be the function computed by the program `s:=sg t`, that is, $s(x)=0$ if $x=0$; otherwise $s(x)=1$. The program `u:=t OR s` can be constructed since $s(s(x)+s(y))$ equals 0 if $x=y=0$, and 1 otherwise. Use `s:=NOT t` and `u:=t OR s` to implement `u:=t AND s` □

Lemma 4 (Arithmetic). *For any array term* $\mathtt{t}^\sigma$ *we have the programs*

- $\{\mathtt{t}=x\}\mathtt{nil}_\sigma\mathtt{(t)}\{\mathtt{t}=0\}$
- $\{\mathtt{t}=x\}\mathtt{suc}_\sigma\mathtt{(t)}\{\mathtt{t}=x+1 \pmod{|\sigma|_b}\}$
- $\{\mathtt{t}=x\}\mathtt{pred}_\sigma\mathtt{(t)}\{\mathtt{t}=x-1 \pmod{|\sigma|_b}\}$

where b is the base of the execution. Each of the programs is of loop rank $\mathrm{lv}(\sigma)\dot{-}1$, *and each of the programs is of the same data rank as the term* t.

Proof. We define the two programs $\mathtt{nil}_\sigma\mathtt{(t)}$ and $\mathtt{suc}_\sigma\mathtt{(t)}$, together with the program $\mathtt{Y:=sg}_\sigma\mathtt{(t)}$, simultaneously by recursion over the build-up of the type σ. The program $\mathtt{Y:=sg}_\sigma\mathtt{(t)}$ will satisfy

$$\{\mathtt{t}^\sigma = x\}\ \mathtt{Y:=sg}_\sigma\mathtt{(t)}\ \{\mathtt{t}^\sigma = x, \mathtt{Y}^{\mathbf{0}} = x'\}$$

where where $x'=0$ if $x=0$; otherwise $x'=1$.

By Lemma 3, we can define the programs when $\sigma\equiv\mathbf{0}$. Now, assume $\sigma\equiv\pi\to\tau$ and that the programs are defined for the types π and τ. Let $\mathtt{nil}_\sigma\mathtt{(t)}\equiv$

$$\mathtt{nil}_\pi\mathtt{(i);\ for}_\pi\,\mathtt{\{\ nil}_\tau\mathtt{(t[i]);\ suc}_\pi\mathtt{(i)\ \}}$$

(Explanation: t represents a $|\pi|_b$-digit number in base $|\tau|_b$. The program sets each digit to 0.) Let $\mathtt{Y:=sg}_\sigma\mathtt{(t)}\equiv$

$$\mathtt{nil}_\pi\mathtt{(i);\ Y:=0;\ for}_\pi\,\mathtt{\{\ Z:=sg}_\tau\mathtt{(t[i]);\ if\ Z\ \{\ Y:=1\ \};\ suc}_\pi\mathtt{(i)\ \}}$$

where Z is a fresh variable of type $\mathbf{0}$. (Explanation: t represents a $|\pi|_b$-digit number in base $|\tau|_b$. The program sets Y to 0 if each digit in t is 0; otherwise Y is set to 1.) Let $\mathtt{suc}_\sigma\mathtt{(t)}\equiv$

$$\mathtt{nil}_\pi\mathtt{(i);\ Y:=1;\ for}_\pi\,\mathtt{\{\ if\ Y\ \{\ suc}_\tau\mathtt{(t[i]);\ Z:=sg}_\tau\mathtt{(t[i]);\ if\ Z\ \{\ Y:=0\ \}\ \}\ \}}$$

where Y and Z are fresh variables of type $\mathbf{0}$. (Explanation: t represents a $|\pi|_b$-digit number in base $|\tau|_b$. The program increases the number held by t[0] by 1 (mod $|\tau|_b$). If the number turns into 0, the program increases the number held by t[1] by 1 (mod $|\tau|_b$). If the number turns into 0, the the number held by t[2] is also modified, and so on.)

This completes the definition of the programs $\mathtt{nil}_\sigma\mathtt{(t)}$ and $\mathtt{suc}_\sigma\mathtt{(t)}$. Let $\mathtt{pred}_\sigma\mathtt{(t)}$ be the program $\mathtt{for}_\sigma\,\mathtt{\{\ suc}_\sigma\mathtt{(t)\ \}}$. We leave to the reader to check that the programs have the properties asserted by the lemma. □

Theorem 2. *(i)* SPACE $2_i^{\text{LIN}} \subseteq \mathcal{I}^{i,i}$ *and (ii)* TIME $2_{i+1}^{\text{LIN}} \subseteq \mathcal{I}^{i,i+1}$.

Proof. First we prove (ii). Let A be a problem in TIME 2_{i+1}^{LIN}. Thus, A is decided by a Turing machine $\mathfrak{m}$ working in time, and thus space, $2_{i+1}^{k|n|}$ for some fixed $k \in \mathbb{N}$. By Lemma 1 we have a type σ of level i such that $|\sigma|_n > 2_{i+1}^{k|n|}$ for all but finitely many values of n. We can w.l.o.g. assume that $\mathfrak{m}$ is a one-way 1-tape Turing machine given by a triple (K, Σ, δ) where

- K is a finite set of states $\{q_0, \ldots, q_v\}$; let q_0 be the $\mathfrak{m}$'s start state, and let q_1 and q_2 be respectively the accept and the reject state
- Σ is a finite alphabet $\{a_0, \ldots, a_w\}$; let a_0 be the blank symbol
- $\delta : K \times \Sigma \to K \times \Sigma \times \{-1, 0, 1\}$ is the transition function, i.e. $\delta(q, a) = (r, b, m)$ means that if $\mathfrak{m}$ scans the symbol a in state q, then $\mathfrak{m}$ will write the symbol b, move the head according to m (-1 is "left", 0 is "no movement", 1 is "right") and proceed to state r.

We extend the transition function such that for all $a \in \Sigma$ we have $\delta(q, a) = (q, a, 0)$ whenever q is the accept or the reject state. For $b \geq v$, we define $C_K : \mathbf{0}_b \to K$ by $C_K(\imath) = q_{\min(\imath,v)}$; for $b \geq w$, we define $C_\Sigma : \Sigma \to \mathbf{0}_b$ by $C_\Sigma(\imath) = a_{\min(\imath,w)}$. For all n such that $n \geq \max(v, w)$ and $|\sigma|_n > 2_{i+1}^{k|n|}$, a configuration of the execution of $\mathfrak{m}$ on input n can be represented by a triple of natural numbers (t, h, s) where

- $t{:}(\sigma \to \mathbf{0})_n$ and $C_\Sigma(t[i]_n)$ is the symbol in the i'th tape cell (for all $i{:}\sigma_n$)
- $h{:}\sigma_n$ and h gives the position of the scanning head
- $s{:}\mathbf{0}_n$ and $C_K(s)$ is the configuration's state.

We define the relation $(t, h, s) \vdash_{\mathfrak{m}} (t', h', s')$ to hold iff $\mathfrak{m}$ in one transition passes from the configuration (t, h, s) to the configuration (t', h', s').

(Claim) There exists a program $\texttt{step}_{\mathfrak{m}}$ such that $(t, h, s) \vdash_{\mathfrak{m}} (t', h', s')$ iff

$$\{\texttt{X}^{\sigma \to \mathbf{0}} = t, \texttt{Y}^\sigma = h, Z^{\mathbf{0}} = s\}\, \texttt{step}_{\mathfrak{m}}\, \{\texttt{X}^{\sigma \to \mathbf{0}} = t', \texttt{Y}^\sigma = h', Z^{\mathbf{0}} = s'\}\ .$$

There exists a program $\texttt{init}_{\mathfrak{m}}$ such that

$$\{\texttt{W}^{\mathbf{0}} = n\}\, \texttt{step}_{\mathfrak{m}}\, \{\texttt{X}^{\sigma \to \mathbf{0}} = t, \texttt{Y}^\sigma = h, Z^{\mathbf{0}} = s\}$$

where (t, h, s) is initial configuration in the execution of $\mathfrak{m}$ on input n. Moreover, both $\texttt{step}_{\mathfrak{m}}$ and $\texttt{init}_{\mathfrak{m}}$ are of loop rank i and data rank $i+1$.

We will not construct the programs $\text{Step}_{\mathfrak{m}}$ and $\text{Init}_{\mathfrak{m}}$ in detail, but we trust the reader to see that the two programs can be constructed from the basic programs given in Lemma 3 together with the programs $\texttt{nil}_\sigma\texttt{(X)}$, $\texttt{suc}_\sigma\texttt{(X)}$, $\texttt{pred}_\sigma\texttt{(X)}$ (needed to "move the head"); $\texttt{X[Y]:=C}_\ell$ for $\ell = 0, \ldots, w$ (needed to "write to the tape"); and a $\texttt{for}_\sigma$-loop (needed to "initialize the tape"). By the lemmas above, all these programs are of are of loop rank less or equal to $\text{lv}(\sigma)$, that is, of loop rank i; and all these programs are of are data rank less or equal to the level of variable $\mathtt{X}^{\sigma\to\mathbf{0}}$ representing the tape, that is, of data rank $i+1$. Hence, (Claim) holds.

Let $\mathtt{tm_m} \equiv \mathtt{init_m}\,;\,\mathtt{for}_\sigma\,\{\,\mathtt{step_m}\,\}$. The program $\mathtt{tm_m}$ is of loop rank i and data rank $i+1$. Further, we have

$$\{\mathtt{W}=n\}\,\mathtt{tm_m}\,\{\mathtt{X}=t,\mathtt{Y}=h,\mathtt{Z}=s\} \qquad (*)$$

where (t,h,s) is the halt configuration in the execution of $\mathtt{m}$ on input n. Given the program $\mathtt{tm_m}$, it is easy to see that we can construct a program $\mathtt{p}$ of the required loop and data rank such that $\mathtt{p}$ accepts n iff $\mathtt{m}$ accepts n. For finitely many values of n, all n less than some fixed number n_0, we tailor $\mathtt{p}$ by hand such that $\mathtt{p}$ accepts n iff $\mathtt{m}$ accepts n for the remaining values of n; that is for all $n \geq n_0$, we construct $\mathtt{p}$ such that n is accepted or rejected depending on whether the halt configuration given by (*) is an accept or a reject configuration. Thus, the problem A is decided by a program of loop rank i and data rank $i+1$, and we have $A \in \mathcal{I}^{i,i+1}$. This completes the proof of (ii)

We turn to the proof of (i). The case SPACE $2_0^{\text{LIN}} \subseteq \mathcal{F}^0$ requires special treatment. We skip the case since the proof of a very similar result can be found in [17]. A few modifications in the proof of (ii) yield a proof of SPACE $2_{i+1}^{\text{LIN}} \subseteq \mathcal{F}^{2(i+1)}$: Assume that the Turing machine $\mathtt{m}$ on input n works in space $2_{i+1}^{k|n|}$ for some fixed $k \in \mathbb{N}$. Then there exists fixed $k' \in \mathbb{N}$ such that $\mathtt{m}$ on input n works in time $2_{i+2}^{k'|n|}$. By Lemma 1 there exists at type σ of level i such that $2_{i+1}^{k|n|} < |\sigma|_n$, and a type π of level $i+1$ such that $2_{i+2}^{k'|n|} < |\pi|_n$. Construct the programs $\mathtt{step_m}$ and $\mathtt{init_m}$ exactly as in the proof of (ii). Then let $\mathtt{tm_m} \equiv \mathtt{init_m}\,;\,\mathtt{for}_\pi\,\{\,\mathtt{step_m}\,\}$, and proceed as in the proof of (ii). The program $\mathtt{p}$ which accepts n iff $\mathtt{m}$ accepts n, will be of loop rank $i+1$ and data rank $i+1$. □

4 The Proof of SPACE $2_i^{\text{LIN}} \subseteq \mathcal{F}^{2i}$ and TIME $2_{i+1}^{\text{LIN}} \subseteq \mathcal{F}^{2i+1}$

Definition. The closed T-term $M : \mathbf{0},\ldots\mathbf{0} \to \mathbf{0}$ *defines* the number-theoretic function f when $M(\overline{a}_1,\ldots,\overline{a}_k) = \overline{f(a_1,\ldots,a_k)}$. The recursion scheme

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

is called *primitive recursion.* (The scheme defines the number-theoretic function f from the number-theoretic functions g and h.) □

Lemma 5 (Primitive recursion). *The set of number-theoretic functions defined by T^--terms of recursor rank 0 is closed under primitive recursion.*

Proof. Let $f(\vec{x},0) = g(\vec{x})$ and $f(\vec{x},y+1) = h(\vec{x},y,f(\vec{x},y))$. Further, let $G : \mathbf{0},\ldots,\mathbf{0} \to \mathbf{0}$ and $H : \mathbf{0},\ldots,\mathbf{0} \to \mathbf{0}$ be T^--terms of recursor rank 0 such that $G(\overline{a}_1,\ldots,\overline{a}_k) = \overline{g(a_1,\ldots,a_k)}$ and $H(\overline{a_1},\ldots,\overline{a_k},\overline{b},\overline{c}) = \overline{h(a_1,\ldots,a_k,b,c)}$. Now, let

$$F \;\equiv\; \lambda\vec{x},y.\mathrm{R}_{\mathbf{0}}(G\vec{x},\lambda yz.(H\vec{x}yz),y)\ .$$

We leave to the reader to verify that $F : \mathbf{0}, \ldots, \mathbf{0} \to \mathbf{0}$ is a T^--terms of recursor rank 0 such that $F(\overline{a}_1, \ldots, \overline{a}_k, \overline{b}) = \overline{f(a_1, \ldots, a_k, b)}$. □

Lemma 6 (Basic functions). *The following number-theoretic functions can be defined by T^--terms of recursor rank 0. (i) 0,1 (constant functions); (ii) P(x) (predecessor); (iii) $x \dot{-} y$ (modified subtraction); (iv) c where $c(x, y_1, y_2) = y_1$ if $x = 0$ and $c(x, y_1, y_2) = y_2$ if $x \neq 0$; (v) f where $f(x, m) = x + 1 \pmod{m+1}$ for $x \leq m$; (vi) $\max(x, y)$.*

Proof. The constant function 1 is defined by the initial T^--term 1. The projection function $u_i^n(x_1, \ldots, x_n) = x_i$ is defined by the T^--term $\lambda x_1 \ldots x_n . x_i$ (for any fixed $i, n \in \mathbb{N}$ such that $1 \leq i \leq n$). The set of functions defined by T^--terms of rank 0, is obviously closed under composition. By Lemma 5 the set is also closed under primitive recursion. Hence, it is sufficient to assure that the functions in the lemma can be defined from projections and the constant 1 by composition and primitive recursion.

To define the constant function 0 is slightly nontrivial. Define g by primitive recursion such that $g(x, 0) = x$ and $g(x, y+1) = y$. Then we can define the predecessor P from g since $P(x) = g(x, x)$. Further, we can define the constant function 0 by $0 = P(1)$. This proves that (i) and (ii) holds. (iii) holds since we have $x \dot{-} 0 = x$ and $x \dot{-} (y+1) = P(x \dot{-} y)$. It is easy to see that (iv) holds. (v) holds since $c(m \dot{-} x, 0, m \dot{-} ((m \dot{-} x) \dot{-} 1)) = x + 1 \pmod m$ for $x \leq m$. (vi) holds since $\max(x, y) = c(1 \dot{-} (x \dot{-} y), x, y)$. □

Lemma 7 (Conditionals). *For any type σ there exists a T^--term $\mathrm{Cond}_\sigma : \mathbf{0}, \sigma, \sigma \to \sigma$ such that $\mathrm{Cond}_\sigma(\overline{n}, F, G) = F$ when $n = 0$, and $\mathrm{Cond}_\sigma(\overline{n}, F, G) = G$ when $n \neq 0$. Moreover, $\mathrm{rk}(\mathrm{Cond}_\sigma) = 0$.*

Proof. We prove the lemma by induction on the structure of σ. Assume $\sigma = \mathbf{0}$. Let $\mathrm{Cond}_{\mathbf{0}} \equiv \lambda x^{\mathbf{0}} y^{\mathbf{0}} z^{\mathbf{0}} . \mathrm{R}_{\mathbf{0}}(y, \lambda u^{\mathbf{0}} v^{\mathbf{0}} . z, x)$. Then

$$\mathrm{Cond}_{\mathbf{0}}(\overline{0}, F, G) \ = \ \mathrm{R}_{\mathbf{0}}(F, \lambda u^{\mathbf{0}} v^{\mathbf{0}} . G, \overline{0}) \ = \ F$$

and

$$\begin{aligned} &\mathrm{Cond}_{\mathbf{0}}(\overline{n+1}, F, G) \ = \\ &\qquad \mathrm{R}_{\mathbf{0}}(F, \lambda u^{\mathbf{0}} v^{\mathbf{0}} . G, \overline{n+1}) \ = \ \lambda u^{\mathbf{0}} v^{\mathbf{0}} . G(\overline{n}, \mathrm{R}_{\mathbf{0}}(F, \lambda u^{\mathbf{0}} v^{\mathbf{0}} . G, \overline{n})) \ = \ G \ . \end{aligned}$$

Assume $\sigma = \tau \to \rho$. Let $\mathrm{Cond}_\sigma \equiv \lambda x^{\mathbf{0}} X^\sigma Y^\sigma z^\tau . \mathrm{Cond}_\rho(x, Xz, Yz)$. Then, by the induction hypothesis, we have $\mathrm{Cond}_\sigma(\overline{0}, F, G) = \lambda z^\tau . \mathrm{Cond}_\rho(\overline{0}, Fz, Gz) = \lambda z^\tau . Fz = F$. The last equality holds since the calculus permits η-reduction. We have $\mathrm{Cond}_\sigma(\overline{n+1}, F, G) = G$ by a similar argument. Assume $\sigma = \tau \times \rho$. Let

$$\mathrm{Cond}_\sigma \equiv \lambda x^{\mathbf{0}} X^\sigma Y^\sigma . \langle \mathrm{Cond}_\tau(x, \mathbf{fst} X, \mathbf{fst} Y), \mathrm{Cond}_\rho(x, \mathbf{snd} X, \mathbf{snd} Y) \rangle \ .$$

Then, by the induction hypothesis, we have $\mathrm{Cond}_\sigma(\overline{n+1}, F, G) =$

$$\langle \mathrm{Cond}_\tau(x, \mathbf{fst} F, \mathbf{fst} G), \mathrm{Cond}_\rho(x, \mathbf{snd} F, \mathbf{snd} G) \rangle = \langle \mathbf{fst} G, \mathbf{snd} G \rangle = G \ .$$

The last equality holds since the calculus permits η-reductions. The proof that $\mathrm{Cond}_\sigma(\overline{0}, F, G) = F$ is similar. Obviously, we have $\mathrm{rk}(\mathrm{Cond}_\sigma) = 0$ for every σ. □

Definition. For $n \in \mathbb{N}$ and terms $M : \sigma \to \sigma$, $N : \sigma$, let $M^n N$ denote the term M repeated n times on N, i.e. $M^{n+1}N = M(M^n N)$ and $M^0 N = N$. □

Lemma 8 (Iterators). *For all types σ and τ there exists a T^--term It^τ_σ : $\mathbf{0}, \tau \to \tau, \tau \to \tau$ such that $\mathrm{It}^\tau_\sigma(\overline{b}, F, G) = F^{|\sigma|_{b+1}}G$. Moreover, we have $\mathrm{rk}(\mathrm{It}^\tau_\sigma) = \mathrm{lv}(\sigma) + \mathrm{lv}(\tau)$.*

Proof. We prove the lemma by induction on the structure of σ.

Assume $\sigma = \mathbf{0}$. Let $\mathrm{It}^\tau_{\mathbf{0}} \equiv \lambda n^{\mathbf{0}} Y^{\tau\to\tau} X^\tau . \mathrm{R}_\tau(Y(X), \lambda x^{\mathbf{0}}.Y, n)$. Obviously, we have $\mathrm{rk}(\mathrm{It}^\tau_{\mathbf{0}}) = \mathrm{lv}(\mathbf{0}) + \mathrm{lv}(\tau)$. We prove by induction on b that $\mathrm{It}^\tau_\sigma(\overline{b}, F, G) = F^{b+1}(G)$. We have $\mathrm{It}^\tau_\sigma(0, F, G) = \mathrm{R}_\tau(F(G), \lambda x.F, 0) = F(G)$. By induction hypothesis we have

$$\begin{aligned}\mathrm{It}^\tau_\sigma(\overline{b+1}, F, G) \;=\; \mathrm{R}_\tau(F(G), \lambda x.F, \overline{b+1}) \;&=\; (\lambda x.F)\overline{b}\mathrm{R}_\tau(F(G), \lambda x.F, \overline{b}) \;= \\ (\lambda x.F)\overline{b}\mathrm{It}^\tau_\sigma(\overline{b}, F, G) \;&=\; FF^{b+1}G \;=\; F^{b+2}G\ .\end{aligned}$$

Thus, the lemma holds when $\sigma = \mathbf{0}$ since $|\mathbf{0}|_{b+1} = b + 1$.

Assume $\sigma = \sigma_1 \times \sigma_2$. Let $\mathrm{It}^\tau_\sigma \equiv \lambda x^{\mathbf{0}} Y^{\tau\to\tau} X^\tau . \mathrm{It}^\tau_{\sigma_1}(x, \mathrm{It}^\tau_{\sigma_2}(x, Y), X)$. Thus,

$$\begin{aligned}\mathrm{rk}(\mathrm{It}^\tau_\sigma) &= \max(\mathrm{rk}(\mathrm{It}^\tau_{\sigma_1}), \mathrm{rk}(\mathrm{It}^\tau_{\sigma_2})) && \text{def. of rk} \\ &= \max(\mathrm{lv}(\sigma_1) + \mathrm{lv}(\tau), \mathrm{lv}(\sigma_2) + \mathrm{lv}(\tau)) && \text{ind. hyp.} \\ &= \max(\mathrm{lv}(\sigma_1), \mathrm{lv}(\sigma_2)) + \mathrm{lv}(\tau) && \\ &= \mathrm{lv}(\sigma_1 \times \sigma_2), +\mathrm{lv}(\tau) && \text{def. of lv} \\ &= \mathrm{lv}(\sigma) + \mathrm{lv}(\tau)\ . && \sigma = \sigma_1 \times \sigma_2\end{aligned}$$

This proves that $\mathrm{rk}(\mathrm{It}^\tau_\sigma)$ has the right recursor rank. Further, we have

$$\begin{aligned}\mathrm{It}^\tau_\sigma(\overline{b}, F, G) &= \mathrm{It}^\tau_{\sigma_1}(\overline{b}, \mathrm{It}^\tau_{\sigma_2}(\overline{b}, F), G) && \text{def. of } \mathrm{It}^\tau_\sigma \\ &= \mathrm{It}^\tau_{\sigma_2}(\overline{b}, F)^{|\sigma_1|_{b+1}}(G) && \text{ind. hyp. on } \sigma_1 \\ &= F^{|\sigma_1|_{b+1} \times |\sigma_2|_{b+1}}(G) && \text{ind. hyp. on } \sigma_2 \\ &= F^{|\sigma|_{b+1}}(G)\ . && \text{def. of } |\sigma|_{b+1}\end{aligned}$$

Assume $\sigma = \sigma_1 \to \sigma_2$. Let $\mathrm{It}^\tau_\sigma \equiv \lambda x^{\mathbf{0}} Y^{\tau\to\tau} X^\tau . (\mathrm{It}^{\tau\to\tau}_{\sigma_1}(x, \mathrm{It}^\tau_{\sigma_2}(x), Y)X)$. We have

$$\begin{aligned}\mathrm{rk}(\mathrm{It}^\tau_\sigma) &= \max(\mathrm{rk}(\mathrm{It}^{\tau\to\tau}_{\sigma_1}), \mathrm{rk}(\mathrm{It}^\tau_{\sigma_2})) && \text{def. of rk} \\ &= \max(\mathrm{lv}(\sigma_1) + \mathrm{lv}(\tau \to \tau), \mathrm{lv}(\sigma_2) + \mathrm{lv}(\tau)) && \text{ind. hyp.} \\ &= \max(\mathrm{lv}(\sigma_1) + \mathrm{lv}(\tau) + 1, \mathrm{lv}(\sigma_2) + \mathrm{lv}(\tau)) && \text{def. of lv} \\ &= \max(\mathrm{lv}(\sigma_1) + 1, \mathrm{lv}(\sigma_2)) + \mathrm{lv}(\tau) && \\ &= \mathrm{lv}(\sigma) + \mathrm{lv}(\tau)\ . && \text{def. of lv}\end{aligned}$$

So, the iterator has the right recursor rank. We will now prove that we indeed have $\mathrm{It}^{\tau}_{\sigma}(\overline{b}, F, G) = F^{|\sigma|_{b+1}}G$. Let $A \equiv (\mathrm{It}^{\tau}_{\sigma_2}\overline{b})$. We prove by induction on k that $(A^k F)G = F^{|\sigma_2|^k_{b+1}}G$ (*). We have $(A^0 F) = F$, and hence $(A^0 F)G = F^{|\sigma_2|^0_{b+1}}G$. Further, we have

$$(A^{k+1}F)G \;=\; (A(A^k F))G \;=\; \mathrm{It}^{\tau}_{\sigma_2}(\overline{b}, A^k F, G) \;=\; (A^k F)^{|\sigma_2|_{b+1}}G \;=\; F^{|\sigma_2|^{k+1}_{b+1}}G.$$

The two last equalities hold by the induction hypothesis on σ_2 and k respectively. This proves (*), and hence

$$\begin{aligned}
\mathrm{It}^{\tau}_{\sigma}(\overline{b}, F, G) &= \mathrm{It}^{\tau\to\tau}_{\sigma_1}(\overline{b}, (\mathrm{It}^{\tau}_{\sigma_2}\overline{b}), F)G && \text{def. of } \mathrm{It}^{\tau}_{\sigma} \\
&= (\mathrm{It}^{\tau}_{\sigma_2}\overline{b})^{|\sigma_1|_{b+1}}F)G && \text{ind. hyp. on } \sigma_1 \\
&= F^{|\sigma_2|_{b+1}^{|\sigma_1|_{b+1}}}G && (*) \\
&= F^{|\sigma|_{b+1}}G\ . && \text{def. of } |\sigma|_{b+1}
\end{aligned}$$

completes the proof of the theorem. □

We will interpret terms as natural numbers. Let $b > 1$, and let M be a closed T-term of type σ. The interpretation $\mathbf{val}_b(M)$ defined below evaluates M to a natural number of type σ_b, that is, to a natural number strictly less than $|\sigma|_b$.

Definition. Let $\mathcal{V}$ be a valuation, that is a set of pairs x/v where x is a variable and $v \in \mathbb{N}$. For any T-term M we define the *value of M at the base b under valuation $\mathcal{V}$*, written $\mathbf{val}^{\mathcal{V}}_b(M)$.

- Let $\mathbf{val}^{\mathcal{V}}_b(0) = 0$; $\mathbf{val}^{\mathcal{V}}_b(1) = 1$; $\mathbf{val}^{\mathcal{V}}_b(x) = v$ if x is a variable and $x/v \in \mathcal{V}$.
- Let $\mathbf{val}^{\mathcal{V}}_b(sM) = \mathbf{val}^{\mathcal{V}}_b(M) + 1 \pmod b$.
- Let $\mathbf{val}^{\mathcal{V}}_b((MN)) = \mathbf{val}^{\mathcal{V}}_b(M)[\mathbf{val}^{\mathcal{V}}_b(N)]_b$.
- Let $\mathbf{val}^{\mathcal{V}}_b(\lambda x^{\sigma} M^{\tau}) = \sum_{i<|\sigma|_b} \mathbf{val}^{\mathcal{V}'}_b(M) \times |\tau|^i_b$ where $\mathcal{V}' = \mathcal{V} \cup \{x/i\}$.
- Let $\mathbf{val}^{\mathcal{V}}_b(\mathbf{fst} M^{\sigma\times\tau}) = \mathbf{val}^{\mathcal{V}}_b(M) \operatorname{div} |\tau|_b$ (integer division).
- Let $\mathbf{val}^{\mathcal{V}}_b(\mathbf{snd} M^{\sigma\times\tau}) = \mathbf{val}^{\mathcal{V}}_b(M) \bmod |\tau|_b$.
- Let $\mathbf{val}^{\mathcal{V}}_b(\langle M^{\sigma}, N^{\tau}\rangle) = \mathbf{val}^{\mathcal{V}}_b(M) \times |\tau|_b + \mathbf{val}^{\mathcal{V}}_b(N)$.
- Recall that R_{σ} has type $\sigma, \mathbf{0} \to \sigma \to \sigma, \mathbf{0} \to \sigma$. Let $\pi = \mathbf{0} \to \sigma \to \sigma, \mathbf{0} \to \sigma$; let $\rho = \mathbf{0} \to \sigma \to \sigma$; let

  $$\mathbf{val}^{\mathcal{V}}_b(\mathrm{R}_{\sigma}) \;\;=\;\; \sum_{u<|\sigma|_b} |\pi|^u_b \;\times\; (\sum_{w<|\rho|_b} |\mathbf{0} \to \sigma|^w_b \times (\sum_{n<|\mathbf{0}|_b} |\sigma|^n_b \times \Upsilon^n))$$

  where $\Upsilon^0 = u$ and $\Upsilon^{n+1} = w[n]_b[\Upsilon^n]_b$.

For any closed term M let $\mathbf{val}_b(M) = \mathbf{val}^{\emptyset}_b(M)$. □

Lemma 9 (Valuation).

(i) *Let M and N be T^--terms of any type such that $M \rhd N$. Fix some $b > 1$. Let $\mathcal{V}$ be a valuation of the free variables in M such that if $\mathcal{V}$ assigns the value v to the variable $x{:}\sigma$, then $v < |\sigma|_b$. Then we have $\mathbf{val}^{\mathcal{V}}_b(M) = \mathbf{val}^{\mathcal{V}}_b(N)$.*

(ii) *Let M and N be closed T-terms of any type such that $M \rhd N$. We have* $\mathbf{val}_b(M) = \mathbf{val}_b(N)$ *for all sufficiently large b.*

(iii) *Let $M : \mathbf{0} \to \mathbf{0}$ be a closed T^--term. We have $M\overline{n} = \overline{m}$ iff $\mathbf{val}_b(M\overline{n}) = m$ for all $b > \max(n, 1)$, and in particular, we have $\mathbf{val}_{\max(n,1)+1}(M\overline{n}) = m$ iff $M\overline{n} = \overline{m}$.*

Proof. The proof of clause (i) and (ii) are straightforward, but very tedious. We skip the details. Clause (iii) follows from clause (i) and (ii) since the reduction process in the calculus T normalizes and $\mathbf{val}_b(\overline{n}) = n$ for any $b > n$. □

Note that clause (ii) of Lemma 9 does not hold for small b, e.g. $\mathbf{fst}\langle\overline{10}, \overline{2}\rangle \rhd \overline{10}$, but $\mathbf{val}_2(\langle\overline{10}, \overline{2}\rangle) = (10 \times 2 + 2) \operatorname{div} 2 = 11$ and $\mathbf{val}_2(\overline{10}) = 0$. The equality $\mathbf{val}_b(\langle\overline{10}, \overline{2}\rangle) = \mathbf{val}_b(\overline{10})$ holds for all $b > 10$.

Lemma 10 (Arithmetic). *For any type σ there exists T^--terms $0_\sigma : \mathbf{0}$, $\mathrm{Suc}_\sigma : \mathbf{0}, \sigma \to \sigma$, $\mathrm{Le}_\sigma : \mathbf{0}, \sigma, \sigma \to \mathbf{0}$, and $\mathrm{Eq}_\sigma : \mathbf{0}, \sigma, \sigma \to \mathbf{0}$ such that*

(i) $\mathbf{val}_{b+1}(0_\sigma) = 0$ *and* $\mathbf{val}_{b+1}(\mathrm{Suc}_\sigma(\overline{b}, F)) = \mathbf{val}_{b+1}(F) + 1 \pmod{|\sigma|_{b+1}}$

(ii) $\mathrm{Le}_\sigma(\overline{b}, F, G) = 0$ *iff* $\mathbf{val}_{b+1}(F) \leq \mathbf{val}_{b+1}(G)$

(iii) $\mathrm{Eq}_\sigma(\overline{b}, F, G) = 0$ *iff* $\mathbf{val}_{b+1}(F) = \mathbf{val}_{b+1}(G)$

for any closed T-terms F and G. Moreover, Suc_σ, Le_σ and Eq_σ have recursor ranks $\leq 2\mathrm{lv}(\sigma)\dot{-}2$ (and 0_σ has recursor rank 0).

Proof. We prove (i), (ii) and (iii) by induction on structure of σ. It follows from Lemma 6 that there are such T^--terms when $\sigma = \mathbf{0}$.

Assume that $\sigma = \pi \to \tau$. Let $0_\sigma \equiv \lambda x^\pi.0_\tau$. Obviously, $\mathrm{rk}(0_\sigma) = 0$. Let $F \equiv$

$$\begin{aligned}
&\lambda b^{\mathbf{0}} X^{\pi\to\tau} Y^{\pi\to\tau} z^{\mathbf{0}\times\pi}. \\
&\qquad \mathrm{Cond}_{\mathbf{0}\times\pi}(\mathrm{Eq}_\tau(b, X(\mathbf{snd}z), Y(\mathbf{snd}z)), \langle\mathbf{fst}z, \mathrm{Suc}_\pi(b, \mathbf{snd}z)\rangle, \\
&\qquad\qquad \mathrm{Cond}_{\mathbf{0}\times\pi}(\mathrm{Le}_\tau(b, X(\mathbf{snd}z), Y(\mathbf{snd}z)), \langle 0_{\mathbf{0}}, \mathrm{Suc}_\pi(b, \mathbf{snd}z)\rangle, \\
&\qquad\qquad\qquad \langle 1, \mathrm{Suc}_\pi(b, \mathbf{snd}z)\rangle))\ .
\end{aligned}$$

Let M, N and j be closed terms. By the induction hypothesis, we have

$$F(\overline{b}, M, N, \langle i, j\rangle) = \begin{cases} \langle i, \mathrm{Suc}_\pi(j)\rangle & \text{if } \mathbf{val}_{b+1}(M)[\mathbf{val}_{b+1}(j)]_{b+1} = \mathbf{val}_{b+1}(N)[\mathbf{val}_{b+1}(j)]_{b+1} \\ \langle 0, \mathrm{Suc}_\pi(j)\rangle & \text{if } \mathbf{val}_{b+1}(M)[\mathbf{val}_{b+1}(j)]_{b+1} < \mathbf{val}_{b+1}(N)[\mathbf{val}_{b+1}(j)]_{b+1} \\ \langle 1, \mathrm{Suc}_\pi(j)\rangle & \text{otherwise.} \end{cases}$$

Hence, (ii) and (iii) hold when $\mathrm{Le}_\sigma \equiv \lambda bXY.\mathbf{fst}\,\mathrm{It}_\pi^{\mathbf{0}\times\pi}(b, F(b, X, Y), \langle 0_{\mathbf{0}}, 0_\pi\rangle)$ and $\mathrm{Eq}_\sigma \equiv \lambda bXY.\mathrm{Cond}_{\mathbf{0}}(\mathrm{Le}_\sigma(b, X, Y), \mathrm{Cond}_{\mathbf{0}}(\mathrm{Le}_\sigma(b, Y, X), 0_{\mathbf{0}}, 1), 1)$. We will now argue that Le_σ and Eq_σ have the required recursor rank. First, we note that $\mathrm{rk}(F) \leq \max(2\mathrm{lv}(\pi)\dot{-}2, 2\mathrm{lv}(\tau)\dot{-}2)$ (*). (Lemma 7 states that $\mathrm{rk}(\mathrm{Cond}_{\mathbf{0}\times\pi}) = 0$, and then (*) follows from the induction hypothesis.) Further,

$$
\begin{aligned}
\mathrm{rk}(\mathrm{Eq}_\sigma) &= \mathrm{rk}(\mathrm{Le}_\sigma) && \text{def. of rk, def. of } \mathrm{Eq}_\sigma \\
&= \max(\mathrm{rk}(F), \mathrm{rk}(\mathrm{It}_\pi^{\mathbf{0}\times\pi})) && \text{def. of rk, def. of } \mathrm{Le}_\sigma \\
&\le \max(\mathrm{rk}(F), \mathrm{lv}(\pi) + \mathrm{lv}(\mathbf{0}\times\pi)) && \text{Lemma 8} \\
&\le \max(2\mathrm{lv}(\pi) \dot{-} 2, 2\mathrm{lv}(\tau) \dot{-} 2, 2\mathrm{lv}(\pi)) && (*) \\
&= \max(2\mathrm{lv}(\tau) \dot{-} 2, 2\mathrm{lv}(\pi)) \\
&= \max(2(\mathrm{lv}(\tau) \dot{-} 1), 2\mathrm{lv}(\pi)) \\
&= 2\max(\mathrm{lv}(\pi), \mathrm{lv}(\tau) \dot{-} 1) \\
&= 2(\max(\mathrm{lv}(\pi)+1, \mathrm{lv}(\tau)) \dot{-} 1) \\
&= 2(\mathrm{lv}(\sigma) \dot{-} 1) && \text{def. of lv, } \sigma = \pi \to \tau \\
&= 2\mathrm{lv}(\sigma) \dot{-} 2\,.
\end{aligned}
$$

Thus, Eq_σ and Le_σ have the required rank. Next we define Suc_σ. Let $a{:}\sigma_b$ and $a'{:}\sigma_b$ be such that $a' = a{+}1 \pmod{|\sigma|_b}$ and $a = v_0|\tau|_b^0 + v_1|\tau|_b^1 + \cdots + v_k|\tau|_b^k$ where $k = |\pi|_b - 1$ and $v_0, \ldots, v_k$ are digits of type τ_b. Then there exists $i \in \{0, \ldots, k\}$ such that

$$a' \;=\; v_0'|\tau|_b^0 \;+\; \cdots \;+\; v_i'|\tau|_b^i \;+\; v_{i+1}|\tau|_b^{i+1} \;+\; \cdots \;+\; v_k|\tau|_b^k$$

where $v_j' = v_j + 1 \pmod{|\tau|_b}$ for $j = 0, \ldots, i$. We call such an i for the *carry border* for the number $a{:}\sigma_b$. Let $\mathrm{C}_\sigma \equiv \lambda b X.\mathbf{snd}\,\mathrm{It}_\pi^{\mathbf{0}\times\pi}(b, G(b,X), \langle 0, 0_\pi\rangle)$ where

$$
\begin{aligned}
G \equiv \lambda b^{\mathbf{0}} X^{\pi\to\tau} z^{\mathbf{0}\times\pi}.\mathrm{Cond}_{\mathbf{0}}(\mathbf{fst}\,z, \mathrm{Cond}_{\mathbf{0}\times\pi}(&\mathrm{Eq}_\tau(b, \mathrm{Suc}_\tau(X(\mathbf{snd}\,z)), 0_\tau), \\
&\langle 0, \mathrm{Suc}_\pi(\mathbf{snd}\,z)\rangle, \langle 1, \mathrm{Suc}_\pi(\mathbf{snd}\,z)\rangle), \langle 1, \mathrm{Suc}_\pi(\mathbf{snd}\,z)\rangle)
\end{aligned}
$$

Then, $\mathbf{val}_{b+1}(\mathrm{C}(\overline{b}, M))$ equals the carry border for $\mathbf{val}_{b+1}(M)$ when $M:\sigma$ is a closed term. Let $\mathrm{Suc}_\sigma \equiv \lambda b^{\mathbf{0}} X^\sigma i^\pi.\mathrm{Cond}_\tau(\mathrm{Le}_\pi(b, i, \mathrm{C}_\sigma(b, X)), \mathrm{Suc}_\tau(X(i)), X(i))$ and (i) holds. By an argument similar to the one showing that the ranks of Eq_σ and Le_σ are bounded by $2\mathrm{lv}(\sigma) \dot{-} 2$, we can show that the rank of C_σ also is bounded by $2\mathrm{lv}(\sigma) \dot{-} 2$. The rank of Suc_σ equals the rank of C_σ.

Assume that $\sigma = \pi \times \tau$. Let $0_\sigma \equiv \langle 0_\pi, 0_\tau\rangle$. Define Suc_σ such that

$$
\mathrm{Suc}_\sigma(b, \langle F, G\rangle) \;=\; \begin{cases} \langle \mathrm{Suc}_\pi(F), \mathrm{Suc}_\tau(G)\rangle & \text{if } \mathrm{Eq}_\tau(b, \mathrm{Suc}_\tau(G)) = 0_\tau \\ \langle F, \mathrm{Suc}_\tau(G)\rangle & \text{otherwise.} \end{cases}
$$

Define Le_σ such that $\mathrm{Le}_\sigma(b, \langle F, G\rangle, \langle F', G'\rangle) = 0$ iff

$$
\begin{aligned}
(\mathrm{Le}_\pi(b, F, F') \;=\; 0 \;\;\wedge\;\; \mathrm{Eq}_\pi(b, F, F') \;>\; 0) \;\vee& \\
&(\mathrm{Le}_\pi(b, G, G') \;=\; 0 \;\;\wedge\;\; \mathrm{Eq}_\pi(b, F, F') \;=\; 0)
\end{aligned}
$$

and Eq_σ such that $\mathrm{Eq}_\sigma(b, F, G) = 0$ iff $\mathrm{Le}_\sigma(b, F, G) = 0 \;\wedge\; \mathrm{Le}_\sigma(b, G, F) = 0$. It is easy to construct T^--terms Suc_σ, Le_σ and Eq_σ with the required properties and recursor ranks. We skip the details. □

Lemma 11 (Modifications). *For any types $\vec{\sigma} = \sigma_1, \ldots, \sigma_k$ and τ there exists a T^--term $\mathrm{Md}_{\vec{\sigma}\to\tau} : \mathbf{0}, \vec{\sigma} \to \tau, \vec{\sigma}, \tau \to \vec{\sigma} \to \tau$ such that*

$$\mathrm{Md}_{\vec{\sigma}\to\tau}(b, F, \vec{G}, V)(\vec{H}) = \begin{cases} V & \text{if } \mathbf{val}_{b+1}(G_i) = \mathbf{val}_{b+1}(H_i) \text{ for } i = 1, \ldots, k \\ F(\vec{H}) & \text{otherwise.} \end{cases}$$

for any closed T-terms $G_i : \sigma_i$ and $H_i : \sigma_i$. Moreover, we have $\mathrm{rk}(\mathrm{Md}_{\vec{\sigma}\to\tau}) \leq 2\max(\mathrm{lv}(\sigma_1), \ldots, \mathrm{lv}(\sigma_k)) \dot{-} 2$ (*).

Proof. We prove the lemma by induction on the length of $\vec{\sigma}$. Assume, the length of $\vec{\sigma}$ equals 1, then $\vec{\sigma} = \rho$ for some type ρ. Let

$$\mathrm{Md}_{\rho\to\tau} \equiv \lambda b^{\mathbf{0}} F^{\rho\to\tau} X^{\rho} V^{\tau} Y^{\rho}.\mathrm{Cond}_{\tau}(\mathrm{Eq}_{\rho}(b, X, Y), V, F(Y)) .$$

Assume, the length of $\vec{\sigma}$ is strictly greater than 1, then $\vec{\sigma} = \rho, \vec{\pi}$. Assume, by the induction hypothesis, that $\mathrm{Md}_{\rho\to\tau}$ and $\mathrm{Md}_{\vec{\pi}\to\tau}$ are defined. Let

$$\begin{aligned}\mathrm{Md}_{\rho,\vec{\pi}\to\tau} \equiv\ & \lambda b^{\mathbf{0}} F^{\rho,\vec{\pi}\to\tau} X_0^{\rho} X_1^{\pi_1} \ldots X_k^{\pi_k} V^{\tau}. \\ & \mathrm{Md}_{\rho\to\tau}(b, F, X_0, \mathrm{Md}_{\vec{\pi}\to\tau}(b, F(X_0), X_1, \ldots, X_k, V)) .\end{aligned}$$

It is easy to see that $\mathrm{rk}(\mathrm{Md}_{\vec{\sigma}\to\tau}) = \max(\mathrm{rk}(\mathrm{Eq}_{\sigma_1}), \ldots, \mathrm{rk}(\mathrm{Eq}_{\sigma_k}), \mathrm{rk}(\mathrm{Cond}_{\tau}))$. Thus, (*) holds by Lemma 7 and Lemma 10. □

Theorem 3. *(i) $\mathcal{I}^{i,i} \subseteq \mathcal{F}^{2i}$ and (ii) $\mathcal{I}^{i,i+1} \subseteq \mathcal{F}^{2i+1}$.*

Proof. Let $\mathtt{p}$ be an imperative program of loop rank i and data rank i' where $i \leq i' \leq i+1$. We will prove that there exists a functional program of recursor rank $i + i'$ deciding the same problem as $\mathtt{p}$, i.e. we will prove that there exists a T^--term $M : \mathbf{0} \to \mathbf{0}$ of recursor rank $i + i'$ such that $M\overline{n} = \overline{0}$ iff $\mathtt{p}$ accepts n. The term M is constructed in details below. From lemmas above, we can conclude that the recursor rank of the term equals the maximum of recursor ranks of the iterator terms introduced by (*) below. Each such term are of the form $\mathrm{It}^{\tau}_{\sigma_1\times\cdots\times\sigma_k\times\mathbf{0}}$ where $\sigma_1, \ldots, \sigma_k$ are the types of the variables occurring in $\mathtt{p}$ and τ is the type of a loop occurring and $\mathtt{p}$. By Lemma 8 the recursor rank of $\mathrm{It}^{\tau}_{\sigma_1\times\cdots\times\sigma_k}$ equals $\mathrm{lv}(\tau) + \max(\mathrm{lv}(\sigma_1), \ldots, \mathrm{lv}(\sigma_k))$, that is $i + i'$. Hence, the term M is of recursor rank $i + i'$.

The remainder of this proof shows how to construct M from the program $\mathtt{p}$.

Let $\mathtt{X}_1^{\sigma_1}, \ldots, \mathtt{X}_k^{\sigma_k}$ denote the variables occurring in $\mathtt{p}$. Let $\mathtt{X}_1$ be the input variable, and thus, $\sigma_1 = \mathbf{0}$. We define the T^--terms $F_1^{\mathtt{p}}, \ldots, F_{k+1}^{\mathtt{p}}$ recursively over the structure of $\mathtt{p}$. The term $F_i^{\mathtt{p}}$ (for $i = 1, \ldots, k$) is of type $\mathbf{0}, \vec{\sigma}, \mathbf{0} \to \sigma_i$ (where $\vec{\sigma} = \sigma_1, \ldots, \sigma_k$), and if

$$\{\mathtt{X}_1 = x_1, \ldots, \mathtt{X}_k = x_k\}\ \mathtt{p}\ \{\mathtt{X}_1 = x_1', \ldots, \mathtt{X}_k = x_k'\}$$

when $\mathtt{p}$ is executed in a sufficiently large base $b+1$, then

$$\mathbf{val}_{b+1}(G_\ell) = x_\ell \text{ for } \ell = 1, \ldots, k \quad\Rightarrow\quad \mathbf{val}_{b+1}(F_j^{\mathtt{p}}(\overline{b}, G_1, \ldots, G_k, y)) = x_j'$$

for any closed T-terms $G_1^{\sigma_1}, \dots, G_k^{\sigma_k}$. Further, we will construct the terms such that `p` accepts n iff $F_{k+1}^{\mathtt{p}}(\overline{\max(n,1)}, \overline{n}, G_2, \dots, G_k, 1)) = 0$ for any closed terms $G_2^{\sigma_2}, \dots, G_k^{\sigma_k}$.

Assume `p` $\equiv$ `q;r`. Let $F_i^{\mathtt{p}} \equiv \lambda b\vec{X}.F_i^{\mathtt{r}}(b, F_1^{\mathtt{q}}(b,\vec{X}), \dots, F_{k+1}^{\mathtt{q}}(b,\vec{X}))$ for $i = 1, \dots, k+1$.

Assume `p` $\equiv$ `if t {q}`. Let $t = X_j$ if `t` $\equiv$ $\mathtt{X}_j$, and let $t \equiv X_j(X_{j_1}, \dots, X_{j_r})$ if `t` $\equiv \mathtt{X}_j[\mathtt{X}_{j_1}, \dots, \mathtt{X}_{j_r}]$. Then, let $F_i^{\mathtt{p}} \equiv \lambda b\vec{X}.\mathrm{Cond}_{\sigma_i}(t, X_i, F_i^{\mathtt{q}}(b,\vec{X}))$ for $i = 1, \dots, k+1$.

Assume `p` $\equiv \mathtt{for}_\tau$ `{q}`. For $j = 1, \dots, k$ let $\mathrm{Pr}_j : \sigma_1 \times \cdots \sigma_k \to \sigma_j$ be a T^--term such that $\mathrm{Pr}_j(\langle Y_1^{\sigma_1}, \dots, Y_k^{\sigma_k}\rangle) = Y_j$. The term Pr_j is easily defined using the deconstructors **fst** and **snd**. Let

$$\begin{aligned} F^{\mathtt{q}} \;\equiv\; & \lambda b^{\mathbf{0}} X^{\sigma_1 \times \cdots \times \sigma_{k+1}}. \\ & \langle F_1^{\mathtt{q}}(b, \mathrm{Pr}_1(X), \dots, \mathrm{Pr}_{k+1}(X)) \,,\, \dots \,,\, F_{k+1}^{\mathtt{q}}(b, \mathrm{Pr}_1(X), \dots, \mathrm{Pr}_{k+1}(X))\rangle \end{aligned}$$

and let

$$F_i^{\mathtt{p}} \;\equiv\; \lambda b\vec{X}.\mathrm{It}^{\tau}_{\sigma_1 \times \cdots \times \sigma_{k+1}}(b, F^{\mathtt{q}}(b), \langle\vec{X}\rangle) \qquad (*)$$

for $i = 1, \dots, k+1$.

Assume `p` $\equiv$ `accept`. Let $F_i^{\mathtt{p}} \equiv \lambda b\vec{X}.X_i$ for $i = 1, \dots, k$. Further, let Zero is be T^--term reducing to $\overline{0}$ (such a term exists by Lemma 6), and let $F_{k+1}^{\mathtt{p}} \equiv \lambda b\vec{X}.\mathrm{Zero}$.

Assume `p` $\equiv$ `t+`. Let S be T^--term such that $S(\overline{x}, \overline{y}) = \overline{y + 1 \pmod{x}}$ (such a term exists by Lemma 6). We split into two subcases.

1. `t` $\equiv \mathtt{X}_m$. Let $F_{k+1}^{\mathtt{p}} \equiv \lambda b\vec{X}.S(b, X_m)$.
2. `t` $\equiv \mathtt{X}_m[\mathtt{X}_{m_1}, \dots \mathtt{X}_{m_r}]$ where $\mathtt{X}_m$ is a variable of type $\tau_1, \dots, \tau_r \to \mathbf{0}$. Let

$$F_m^{\mathtt{p}} \;\equiv\; \lambda b\vec{X}.\mathrm{Md}_{\vec{\tau} \to \mathbf{0}}(b, X_m, X_{m_1}, \dots, X_{m_r}, S(b, X_m(X_{m_1}, \dots, X_{m_r}))) \,.$$

For both subcases let $F_i^{\mathtt{p}} \equiv \lambda b\vec{X}.X_i$ for $i \neq m$. This completes the construction of the terms $F_1^{\mathtt{p}}, \dots, F_{k+1}^{\mathtt{p}}$.

Let G be a T^--term such that $G(\overline{x}) = \overline{\max(x,1)}$ (such a term exists by Lemma 6), let G_j be any closed term of type σ_j, and finally, let

$$M \;\equiv\; \lambda x.F_{k+1}^{\mathtt{p}}(G(x), x, G_2, \dots, G_k, 1)$$

□

5 The proof of $\mathcal{F}^{2i} \subseteq$ SPACE 2_i^{LIN} and $\mathcal{F}^{2i+1} \subseteq$ TIME 2_{i+1}^{LIN}

Definition. For any $n \in \mathbb{N}$, the $\lambda\langle\rangle_n$-calculus is the standard typed λ-calculus extended with the constants $0 : \mathbf{0}$ (zero), $s : \mathbf{0} \to \mathbf{0}$ (successor) and the *sequence* of type $\mathbf{0} \to \mathbf{0}$ of length $n+1$, that is, if $M_0, \dots, M_n$ are terms of type $\mathbf{0}$, then

$\langle M_0, \ldots, M_n \rangle$ is a term of type $\mathbf{0} \to \mathbf{0}$. In addition to the standard reduction rules for the typed λ-calculus, the $\lambda\langle\rangle_n$-calculus has the reduction rule

$$\langle M_0, \ldots, M_n \rangle\, \overline{i} \;\rhd\; M_i \,. \qquad (\langle\rangle\text{-reduction})$$

The next couple of definitions are valid for both T-terms and $\lambda\langle\rangle_n$-terms. The *term rank* $\mathrm{rk}_t(M)$ of the term M is defined as the least $n \in \mathbb{N}$ such that for any subterm $N\!:\!\sigma$ of M we have $\mathrm{lv}(\sigma) \leq n$. Further, let $\#M$ denote the length of the term M. (Any reasonable definition of length will do, we may for instance count the number of symbols in M.)

Let $\textsc{rep}(M)$ denote the encoding of the $\lambda\langle\rangle_n$-term M in a finite alphabet, and let $|\textsc{rep}(M)|$ denote the length of the encoding $\textsc{rep}(M)$. The encoding should be such that we have $|\textsc{rep}(M)| \leq p(\#M)$ for some polynomial p. (Any reasonable encoding will satisfy this requirement. The encoding is needed since Turing machines work with finite alphabets, whereas $\lambda\langle\rangle_n$-terms are terms over an infinite alphabet.) $\square$

Lemma 12 (Embedding). *Let $M\!:\!\mathbf{0} \to \mathbf{0}$ be a fixed and closed T^--term. Then, there exists a $\lambda\langle\rangle_n$-term $\tilde{M}$ such that (i) $\mathrm{rk}_t(\tilde{M}) = \mathrm{rk}(M) + 1$ and (ii) $M\overline{n} = \overline{m}$ iff $\tilde{M} = \overline{m}$. Moreover, (iii) given the number n as input, a Turing machine can generate $\textsc{rep}(\tilde{M})$ in time $2^{c|n|}$ where c is some fixed number. (The number c will depend on the term M, but M is fixed.)*

Proof. For each $n \in \mathbb{N}$ we define a mapping Γ_n of the T-terms into the $\lambda\langle\rangle_n$-terms. Let $\Gamma_n(x) = x$ if x is a variable; $\Gamma_n(0) = 0$; $\Gamma_n(1) = s0$; $\Gamma_n(sM) = s\Gamma_n(M)$; $\Gamma_n(\lambda x M) = \lambda x \Gamma_n(M)$; $\Gamma_n((MN)) = (\Gamma_n(M)\Gamma_n(N))$; $\Gamma_n(\langle M, N\rangle) = \langle \Gamma_n(M), \Gamma_n(N)\rangle$; $\Gamma_n(\mathbf{fst} M) = \mathbf{fst}\Gamma_n(M)$; $\Gamma_n(\mathbf{snd} M) = \mathbf{snd}\Gamma_n(M)$; and

$$\begin{aligned}\Gamma_n(\mathrm{R}_\sigma) \;=\; & \lambda Z^\sigma Y^{\mathbf{0}\to\sigma\to\sigma} X^{\mathbf{0}} \vec{x}. \\ & \langle\, Z\vec{x}\,,\, Y(\overline{0}Z)\vec{x}\,,\, Y(\overline{1}(Y\overline{0}Z))\vec{x}\,,\, \ldots\,,\, Y(\overline{n-1}\ldots Y(\overline{0}Z)\ldots)\vec{x}\,\rangle\end{aligned}$$

where $\sigma = \tau_1, \ldots, \tau_k \to \mathbf{0}$ and $\vec{x} = x_1^{\tau_1}, \ldots, x_k^{\tau_k}$. (Any type σ has the form $\tau_1, \ldots, \tau_k \to \mathbf{0}$ for some $\tau_1, \ldots, \tau_k$. If $\sigma = \mathbf{0}$, then $k = 0$.)

(a) Let N be the $\beta\eta$-normal form of the term $M\overline{n}$. Then $\mathrm{rk}_t(N) = \mathrm{rk}(N) + 1$.
(b) We have $\Gamma_n(\mathrm{R}_\sigma)GF\overline{j} = F(\overline{j-1}\ldots F(\overline{1}(F\overline{0}G))\ldots)$ for any $j \leq n$.
(c) For any term M there exists a polynomial p such that $\#\Gamma_n(M) \leq p(n)$.

We prove (a). Obviously, it cannot be the case that $\mathrm{rk}_t(N) < \mathrm{rk}(N) + 1$. Suppose that $\mathrm{rk}_t(N) > \mathrm{rk}(N) + 1$. Then there exists a subterm $N'\!:\!\rho \to \tau$ of N such that $\mathrm{rk}_t(N') = \mathrm{rk}_t(N) = \mathrm{lv}(\rho \to \tau) > \mathrm{rk}(N) + 1$. First we note that N' cannot be a variable. (Because N is a closed term, and if N' were a variable, we would have $\mathrm{rk}_t(N) > \mathrm{lv}(\rho \to \tau)$.) Thus, N' has the form $\lambda x^\rho . P^\tau$, and since N has type $\mathbf{0}$ there will be a subterm of N on the form $\lambda x^\rho . P^\tau Q^\rho$. This contradicts that N is in the $\beta\eta$-normal form. This proves (a). Further, we have

$$\Gamma_n(\mathrm{R}_\sigma)GF\overline{j} \;=\; \lambda\vec{x}.(F(\overline{j-1}\ldots F(\overline{0}G)\ldots)\vec{x}) \;=\; F(\overline{j-1}\ldots F(\overline{1}F(\overline{0}G))\ldots)\,.$$

The first equality holds by three β-reduction and one $\langle\rangle$-reduction; the second equality holds by k η-reductions. Hence, we see that (b) holds. It easy to see that (c) holds by inspecting the definition of Γ_n.

Let N be the $\beta\eta$-normal form of the term $M\overline{n}$, and let $\tilde{M}$ be the term $\Gamma_n(N)$. It follows from respectively (a) and (b) that respectively (i) and (ii) hold. Further, it is easy to see that $\#N$ is bounded by a polynomial in n, and then by (c), $\#\tilde{M}$ is bounded by a polynomial in n. Our encoding scheme for terms guarantees that also $|\textsc{rep}(\tilde{M})|$ will be bounded by a polynomial in n. It is easy to see that there exists a polynomial p such that $\textsc{rep}(\tilde{M})$ can be generated by a Turing machine in no more than $p(|\textsc{rep}(\tilde{M})|)$ steps. This entails that there exists a fixed $c \in \mathbb{N}$ such that $\textsc{rep}(\tilde{M})$ can be generated in time $2^{c|n|}$. Hence, (iii) holds. $\square$

Definition. We extend the definition of $\mathbf{val}_b$ given at page 14 by

$$\mathbf{val}_b^{\mathcal{V}}(\langle M_0, \ldots, M_n\rangle) \;=\; \sum_{i<n+1} \mathbf{val}_b^{\mathcal{V}}(M_i) \times |\mathbf{0}|_b^i \;.$$

(Thus, $\mathbf{val}_{n+1}(M)$ is defined for any closed $\lambda\langle\rangle_n$-term M.) $\square$

Lemma 13. *Let $M:\mathbf{0}$ be a closed $\lambda\langle\rangle_n$-term of term rank $k+2$.*

(i) There exist a polynomial p and a constant $c \in \mathbb{N}$ such that $\mathbf{val}_{n+1}(M)$ can be computed by a Turing machine in space $p(|\textsc{rep}(M)|) \times 2_k^{c|n|}$.
(ii) There exist a polynomial p and a constant $c \in \mathbb{N}$ such that $\mathbf{val}_{n+1}(M)$ can be computed by a Turing machine in time $p(|\textsc{rep}(M)|) \times 2_{k+1}^{c|n|}$.

Proof. In this proof we will call a redex *maximal* if it is of the form $(\lambda x^\sigma.P^\tau)Q^\sigma$ where $\mathrm{lv}(\sigma \to \tau) = k+2$. We also need the notion of a *semi-reduction.* In a semi-reduction we do not replace a variable by a term as we do in an ordinary β-reduction. Instead we store the term somewhere else and replace the variable by an address (pointer) to the storage location. The Turing machine constructed below saves space by using such a strategy.

Let $M_{p_i}^x$ denote the term we get when each occurrence the variable x in M is replaced by the *pointer* p_i. (We use $p_1, p_2, p_3 \ldots$ to denote pointers. The Turing machine constructed below uses binary numbers to represent the pointers.) We will say that a string on the form

$$C\{(\lambda x.P)Q\}/p_1 : M_1/p_2 : M_2/\ldots/p_\ell : M_\ell$$

(where $C\{(\lambda x.P)Q\}$ is a representation of a $\lambda\langle\rangle_n$-term possibly containing pointers with maximal redex $(\lambda x.P)Q$, and where $M_1, \ldots, M_\ell, P, Q$ are representations of $\lambda\langle\rangle_n$-terms possibly containing pointers) *semi-reduces* to the string

$$C\{P_{p_{\ell+1}}^x\}/p_1 : M_1/p_2 : M_2/\ldots/p_\ell : M_\ell/p_{\ell+1} : Q \;.$$

Let $M:\mathbf{0}$ be the closed $\lambda\langle\rangle_n$-term of term rank $k+2$ given in the lemma. We will construct a Turing machine computing $\mathbf{val}_{n+1}(M)$. Let w_0 be $\textsc{rep}(M)$. The Turing machine starts with w_0 on its work tape, picks a maximal redex in w_0

and semi-reduces w_0 to w_1. It will pick the leftmost maximal redex (λxPR) such that there are no maximal redexes inside R. Thereafter it semi-reduces w_1 to w_2 following the same procedure, then w_2 to w_3, and so on. Sooner or later, say after $\jmath$ steps, the process will terminate since there will be no maximal redexes left. The string $w_\jmath$ has the form $P/p_1 : M_1/p_2 : M_2/\dots/p_\jmath : M_\jmath$ and represents a $\lambda\langle\rangle_n$-term of term rank $k+1$. (Note that no semi-reductions will take place inside the terms $M_1, \dots, M_\jmath$ since these terms do not contain maximal redexes.) The Turing machine can move freely back and forth in the represented term by following the pointers and pushing the return addresses on a stack. It should be obvious that we could construct the Turing machine such that

> the number of steps the Turing machine needs to generate $w_\jmath$ from the input $\textsc{rep}(M)$ is bounded by $p(|\textsc{rep}(M)|)$ for some polynomial p. (*)

Let Q be the term of term rank $k+1$ represented by $w_\jmath$. We have $\mathbf{val}_{n+1}(Q) = \mathbf{val}_{n+1}(M)$. After generating $w_\jmath$, the Turing machine will compute $\mathbf{val}_{n+1}(Q)$ using *registers*. A register is nothing but a marked area on one of the tapes dedicated to store a natural number. The function $\mathbf{val}^{\mathcal{V}}_{n+1}$ is defined recursively over the structure of $\lambda\langle\rangle_n$-terms. The Turing machine will compute the value $\mathbf{val}^{\mathcal{V}}_{n+1}(S)$ of a composed term S by computing the values of its subterms, store the results away in registers, and then retrieve the results when they are needed in the computation of $\mathbf{val}^{\mathcal{V}}_{n+1}(S)$. E.g. to compute the value $\mathbf{val}^{\mathcal{V}}_{n+1}((\lambda x.S)R)$, the Turing machine will first compute $a = \mathbf{val}^{\mathcal{V}}_{n+1}(\lambda xS)$, store a in a register, thereafter compute $b = \mathbf{val}^{\mathcal{V}}_{n+1}(R)$, store b in a register, and finally compute the value $\mathbf{val}^{\mathcal{V}}_{n+1}((\lambda x.S)R)$ by computing the number $a[b]_{n+1}$. The Turing machine will only depart from this natural recursive procedure when it encounters subterms of level $k+1$. There will be some such subterms since Q is of term rank $k+1$. For the sake of the argument assume that such a subterm has form λxR where $\mathrm{rk}_t(R) = k$. In such a case the Turing machine will compute the value $\mathbf{val}^{\mathcal{V}}_{n+1}((\lambda x.R)S)$ by first computing $a = \mathbf{val}^{\mathcal{V}}_{n+1}(S)$ and then compute the value $\mathbf{val}^{\mathcal{V}'}_{n+1}(R)$ where $\mathcal{V}' = \mathcal{V} \cup \{x/a\}$. It should be obvious that it is possible to design the Turing machine such that it never will store numbers of type σ where $\mathrm{lv}(\sigma) = k+1$.

We will now argue that the Turing machine sketched above works within the space and time constrains stated in the lemma. First the Turing machine generates $w_\jmath$, and by (*), it needs no more than $p(|\textsc{rep}(M)|)$ tape cells to do so. Then the Turing machine computes $\mathbf{val}_{n+1}(M)$ by using registers. The greatest number stored in a register during the computation is bounded by $|\xi|_{n+1}$ where ξ is of level k. Lemma 2 says there exists a constant $c_1 \in \mathbb{N}$ such that $|\xi|_{n+1} \leq 2^{c_1|n|}_{k+1}$. The Turing machine represents the numbers in the registers binary, and hence the number of tape cells required for one register is bounded by $2^{c_2|n|}_k$ for some constant $c_2 \in \mathbb{N}$. The total number of registers required will be bounded by the length of $w_\jmath$, and thus, the total number of tape cells required will be bounded by $p(|\textsc{rep}(M)|) \times 2^{c|n|}_k$ for some polynomial p and some $c \in \mathbb{N}$. This proves (i). The proof of (ii) is similar. □

Lemma 14 (Rank Reduction). *Let $M:\mathbf{0}$ be a closed $\lambda\langle\rangle_n$-term of term rank $k+2$. (i) There exists a $\lambda\langle\rangle_n$-term N of term rank $k+1$ such that $M = N$ and $\#N \leq 2^{c\#M}$ for some fixed $c \in \mathbb{N}$. Moreover, (ii) given* $\textsc{rep}(M)$ *as input, a Turing machine can generate* $\textsc{rep}(N)$ *in time* $2^{c|\mathrm{REP}(M)|}$ *for some fixed* $c \in \mathbb{N}$.

Proof. The proof of (i) is similar to the standard proofs found in the literature for eliminating "cuts" in the typed λ-calculus. See Schwictenberg [26] and Beckmann [3]. Since each sequence is of type $\mathbf{0} \to \mathbf{0}$, there will be no $\langle\rangle$-reductions involved in the rank reduction process from rank $k+2$ to rank $k+1$.

We argue that also (ii) holds. The rank reduction process might introduce new variables (in order to avoid that β-reductions result in name conflicts). Still, the number of variables in N will be bounded by $2^{c\#M}$, and thus, using a reasonable encoding scheme, we have $|\textsc{rep}(N)| \leq 2^{c'|\mathrm{REP}(M)|}$ for some $c' \in \mathbb{N}$. The number of steps required to generate $\textsc{rep}(N)$ is bounded by $p(|\textsc{rep}(N)|)$ for some polynomial p, and thus also bounded by $2^{c|\mathrm{REP}(M)|}$ for some $c \in \mathbb{N}$. $\square$

Theorem 4. *(i)* $\mathcal{F}^{2i} \subseteq \text{SPACE } 2_i^{\text{LIN}}$ *and (ii)* $\mathcal{F}^{2i+1} \subseteq \text{TIME } 2_{i+1}^{\text{LIN}}$.

Proof. We prove (i). The proof splits into the cases $i = 0$ and $i > 0$.

Case $i > 0$. Then $i = k+1$ and $2i = 2k+2$ for some $k \in \mathbb{N}$. Let $M:\mathbf{0} \to \mathbf{0}$ be a closed T^--term of rank $2k+2$. We will prove that there exists a Turing machine which on input n works in space $2_{k+1}^{c|n|}$ for some constant $c \in \mathbb{N}$, and computes the number m such that $M\overline{n} = \overline{m}$.

First, the Turing machine generates the representation $\textsc{rep}(\tilde{M})$ of the $\lambda\langle\rangle_n$-term $\tilde{M}$ given in Lemma 12. The term $\tilde{M}$ is of term rank $2k+3$, and $\textsc{rep}(\tilde{M})$ is generated in time $2^{c_0|n|}$ for some fixed $c_0 \in \mathbb{N}$. Thereafter, the Turing machine generates the representation $\textsc{rep}(P)$ of a $\lambda\langle\rangle_n$-term P of term rank $k+3$ such that $P = \tilde{M}$. By Lemma 14, we have $c_1 \in \mathbb{N}$ such that $\textsc{rep}(P)$ can be generated from $\textsc{rep}(\tilde{M})$ in time $2_k^{c_1|\mathrm{REP}(\tilde{M})|}$. Thus, there exists $c_2 \in \mathbb{N}$ such that $\textsc{rep}(P)$ is generated from n in time (and hence also in space) $2_{k+1}^{c_2|n|}$. Finally, the Turing machine computes the value $\mathbf{val}_{n+1}(P)$. By Lemma 13 it is possible for a Turing machine to complete this task in space $p(\textsc{rep}(P)) \times 2_{k+1}^{c_3|n|}$ where p is a polynomial and $c_3 \in \mathbb{N}$. Note that $p(\textsc{rep}(P)) \times 2_{k+1}^{c_3|n|} \leq p(2_{k+1}^{c_2|n|}) \times 2_{k+1}^{c_3|n|} \leq 2_{k+1}^{c|n|}$ holds for some fixed $c \in \mathbb{N}$.

Case $i = 0$. Let $M:\mathbf{0} \to \mathbf{0}$ be a closed T^--term of rank 0. Let $x:\mathbf{0}$ be a variable not occurring in M, and let P be Mx in the $\beta\eta$-normal form. A Turing machine can compute $\mathbf{val}_{n+1}(P[x := \overline{n}])$ using k_0 registers holding numbers of type $\mathbf{0}$. (A register is nothing but a marked area of the tape.) The number k_0 of registers required is determined by P, and thus independent of n. Each register will hold a number in the set $\mathbf{0}_{n+1} = \{0, \ldots, n\}$. Thus, the number m such that $\overline{m} = M\overline{n}$ can be computed in space $c|n|$ for some fixed $c \in \mathbb{N}$. This completes the proof of (i)

We prove (ii). Let $M:\mathbf{0} \to \mathbf{0}$ be a closed T^--term of rank $2i+1$. We will prove that there exists a Turing machine which on input n works in time $2_{i+1}^{c|n|}$ for some constant $c \in \mathbb{N}$, and computes th number m such that $M\overline{n} = \overline{m}$.

The Turing machine works similarly to the Turing machine in the case $i > 0$ above. First it spends $2_{i+1}^{c_1|n|}$ steps to generate the representation of a $\lambda\langle\rangle_n$-term P of term rank $i+2$ such that $\mathbf{val}_{n+1}(P) = \mathbf{val}_{n+1}(M\overline{n})$. By Lemma 13 there exist a polynomial p and $c_2 \in \mathbb{N}$ such that the Turing machine need no more than $p(|\text{REP}(P)|) \times 2_{i+1}^{c_2|n|}$ steps to compute $\mathbf{val}_{n+1}(P)$. It follows that the Turing machine on input n computes the number m in time $2_{i+1}^{c|n|}$ for some fixed $c \in \mathbb{N}$. □

6 Another Alternating Space-Time Hierarchy

Let LOGSPACE denote the set of problems decided by a Turing machine working in logarithmic space. Let denote TIME 2_i^{POL} (SPACE 2_i^{POL}) the set of problems decided by a Turing machine working in time (space) $2_i^{p(|x|)}$ for some polynomial p. Then we have an alternating space-time hierarchy

$$\text{LOGSPACE} \subseteq \text{TIME } 2_0^{\text{POL}} \subseteq \text{SPACE } 2_0^{\text{POL}} \subseteq \text{TIME } 2_1^{\text{POL}} \subseteq \text{SPACE } 2_1^{\text{POL}} \subseteq \text{TIME } 2_2^{\text{POL}} \subseteq \ldots$$

analogous to the hierarchy studied in the precedent sections. The analogous open problems do also emerge. Let $\mathcal{C}_i$, $\mathcal{C}_{i+1}$, $\mathcal{C}_{i+2}$ be three arbitrary consecutive classes in the hierarchy. It is well-known that $\mathcal{C}_i \subset \mathcal{C}_{i+2}$, so at least one of the two inclusions $\mathcal{C}_i \subseteq \mathcal{C}_{i+1}$ and $\mathcal{C}_{i+1} \subseteq \mathcal{C}_{i+2}$ must be strict. Still, for any fixed $j \in \mathbb{N}$, it is an open problem if $\mathcal{C}_j$ is strictly included in $\mathcal{C}_{j+1}$. Note that TIME 2_0^{POL} and SPACE 2_0^{POL} are the classes usually denoted respectively P and PSPACE in the literature, so the notorious open problem

$$\text{LOGSPACE} \overset{?}{\subset} \text{P} \overset{?}{\subset} \text{PSPACE}$$

emerge at the bottom of the hierarchy.

The relationship between the two alternating space-time hierarchies is also a bit of a mystery. The only thing known about the relationship between SPACE 2_i^{LIN} and TIME 2_i^{POL} is that the two classes cannot be equal. So, it is known that e.g. LINSPACE $\neq$ P, but do we have LINSPACE $\subset$ P or P $\subset$ LINSPACE? Well, most complexity theorists believe that the two classes are incomparable, i.e. that neither of them is included in the other, but no proof exists.

Programming languages which are slight modifications of the imperative and the functional language defined in Section 2 capture the complexity-theoretic hierarchy under discussion. These are variants of the languages where the programs receive the input represented binary notation. An imperative program receives the input $x \in \mathbb{N}$ in a variable `X` of type $\mathbf{0} \to \mathbf{0}$. When the execution starts the array `X` will store the (shortest possible) binary representation of x digit by digit, and the execution base b is set to $b = |x| + 1$. Otherwise, the imperative language is defined exactly as in Section 2. In the functional language a binary notation for the natural numbers can be introduced straightforwardly by two successor constants $s_0 : \mathbf{0} \to \mathbf{0}$ and $s_1 : \mathbf{0} \to \mathbf{0}$. We will then need to adjust the recursor accordingly. Alternatively, we can leave the syntax of the functional language unaltered, and let the input $x \in \mathbb{N}$ be given to the program by a term $N : \mathbf{0} \to \mathbf{0}$ where $N\overline{k}$ normalize to the numeral identifying the k'th digit in the binary representation of x. The latter option requires that the length of the input is given to the program as a numeral. Otherwise, the functional language is defined exactly as in Section 2.

Let us say that programs receiving the input the way just prescribed are *binary mode* working programs; whereas programs receiving the input the way prescribed in Section 2 are *unary mode* working programs. We can now proceed as in Section 2 and define $\hat{\mathcal{I}}^{i,j}$ as the class of problems decided by a binary mode working imperative program of loop rank i and data rank j. (So $\mathcal{I}^{i,j}$ is the corresponding class for unary mode working programs.) Further, we define $\hat{\mathcal{F}}^i$ as the class of problems decided by a binary mode working functional program of recursor rank i. (So $\mathcal{F}^i$ is the corresponding class for unary mode working programs.)

Theorem 5. *We have* $\text{SPACE } 2_i^{\text{LIN}} = \hat{\mathcal{I}}^{i+1,i+1} = \hat{\mathcal{F}}^{2i+2}$ *and* $\text{TIME } 2_i^{\text{LIN}} = \hat{\mathcal{I}}^{i,i+1} = \hat{\mathcal{F}}^{2i+1}$ *for any* $i \in \mathbb{N}$. *Moreover, we have* $\text{LOGSPACE} = \hat{\mathcal{I}}^{0,0} = \hat{\mathcal{F}}^0$

We have not carried out all the technical details in the proof of Theorem 5, but we have carried out quite a few details, and it is definitely possible to prove Theorem 5 along the same lines as we have proved Theorem 1. The two proof are essentially the same. Thus, we see that all the major deterministic complexity classes are captured by fragments of our programming languages. Table 1 gives a summary.

Table 1. The table gives a summary of our results in a fairly standard nomenclature.

Imperative programs		**Input mode**		**Functional programs**
Loop rank	Data rank	binary	unary	Recursor rank
0	0	LOGSPACE	LINSPACE	0
0	1	P	EXP	1
1	1	PSPACE	EXPSPACE	2
1	2	POLYEXP	TIME 2_2^{LIN}	3
2	2	POLYEXPSPACE	SPACE 2_2^{LIN}	4
$\vdots$	$\vdots$	$\vdots$	$\vdots$	
i	$i+1$	TIME 2_i^{POL}	TIME 2_{i+1}^{LIN}	2i + 1
$i+1$	$i+1$	SPACE 2_i^{POL}	SPACE 2_{i+1}^{LIN}	2i + 2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	

7 Discussion and References

7.1 Finite Model Theory and the Goerdt-Seidl Hierarchy

Goerdt-Seidl [11] (and Goerdt [10]) use finite models to characterize the alternating space-time hierarchy presented in Section 6. The inclusions $\mathcal{F}^{2i} \subseteq \text{SPACE } 2_i^{\text{LIN}}$ and $\mathcal{F}^{2i+1} \subseteq \text{TIME } 2_{i+1}^{\text{LIN}}$ can be obtained directly from the theorems in [11]. We hope that our proofs of the inclusions in Section 5 are more enlightening and transparent than the corresponding proofs in [11]. (Although it will be unfair to say that the proofs in Section 5 are substantially different from the ones in [11]). The inclusions $\text{SPACE } 2_i^{\text{LIN}} \subseteq \mathcal{F}^{2i}$ and $\text{TIME } 2_{i+1}^{\text{LIN}} \subseteq \mathcal{F}^{2i+1}$ do not follow from Goerdt and Seidl's work.

7.2 Implicit Characterizations of Complexity Classes

The characterizations of the complexity classes given by the two main theorems of this paper are so-called *implicit* characterizations, that is, they contain no references to explicit resource bounds. To get rid of explicit resource bound and obtain implicit characterizations of complexity classes, so-called *ramification techniques* (also known as tiering techniques) have shown to be successful. Numerous examples of ramification can be found in the literature, Simmons [28] may be the first, Bellantoni and Cook's [5] distinction between normal and safe variables is probably the most known. In particular, ramification techniques are used to restrict higher type recursion to the Kalmár elementary level, e.g. Simmons [29], Leivant [20], Beckmann and Weiermann [4]. By using so-called linearity constraints in addition to ramification, higher type recursion can be restricted further down to the "polytime" level, e.g. Bellantoni-Niggl-Schwichtenberg [7], Schwichtenberg-Bellantoni [27]. The *measure* techniques, which go back to Niggl and are used e.g. in Bellantoni-Niggl [6] and Kristiansen-Niggl [16], have a different flavor than the ramification techniques. Still, with the respect to characterizing complexity classes, measure techniques are in some sense ramification techniques in disguise, in other respects, e.g. with respect to (automatic) complexity analysis of programs, one can say that measure techniques have advantages in comparison with ramification techniques. (For more on measure techniques and Niggl's work see Niggl [22].) In this paper we achieve implicit characterizations of complexity classes by simply removing successor-like functions from a standard computability-theoretic framework. This technique is qualitatively different from the techniques mentioned above. Jones [14] [13] uses the same technique (with functional programming languages). So do the authors in [18] (with functionals of higher types), [17] (with imperative programming languages) and [15] (with function algebras).

7.3 Product Types and the Small Grzegorczyk Classes

If we had left out explicit pairing and product types from the functional language, we could still prove that the $\mathcal{F}$-hierarchy matches the alternating space-time hierarchy level by level with one exception. When explicit pairing is available, we can prove that $\mathcal{F}^0 = \textsc{linspace}$; when it is not available, all we can prove is that $\mathcal{F}^0 = \mathcal{E}^0_*$ where $\mathcal{E}^0$ is the 0'th Grzegorczyk class. It is known that $\textsc{linspace} = \mathcal{E}^2_*$ where $\mathcal{E}^2$ is the seconds Grzegorczyk class (Ritchie [24]); it is not known if any of the inclusions $\mathcal{E}^0_* \subseteq \mathcal{E}^1_* \subseteq \mathcal{E}^2_*$ are strict. Barra-Kristiansen [2] relate this open problem to the research presented in this paper. They introduce a hierarchy $\mathcal{L}^0_\star \subseteq \mathcal{L}^1_\star \subseteq \mathcal{L}^2_\star \subseteq \ldots$ where $\mathcal{L}^0_\star = \mathcal{E}^0_*$ and $\bigcup_{i\in\mathbb{N}} \mathcal{L}^i_\star = \mathcal{E}^2_*$. The classes in the hierarchy are induced by fragments of the typed λ-calculus extended by recursors and the constant 1. For more on the small Grzegorczyk classes see Chapter 5 in Rose [25], Kutylowski [19], and Barra-Kristiansen [2].

References

1. J. Avigad and S. Feferman. *Gödel's functional interpretation.* In "Handbook of Proof Theory" Buss (Ed), Elsevier, 1998.
2. G.M. Barra and L. Kristiansen. *The small Grzegorczyk classes and the typed λ-calculus.* Accepted for publication in the proceedings from CiE'05, Cooper, Löwe, Torenvliet (eds.), Springer LNCS.
3. A. Beckmann. *Exact bounds for length reductions in typed λ-calculus.* The Journal of Symbolic Logic 66, No.3 (2001), 1277-1285.
4. A. Beckmann and A. Weiermann. *Characterizing the elementary recursive functions by a fragment of Gödel's T.* Arch. Math. Logic 39, No.7 (2000), 475-491.
5. S. J. Bellantoni and S. Cook. *A New Recursion-Theoretic Characterization of the Polytime Functions.* Computational Complexity, 2 (1992), 7–110.
6. S. J. Bellantoni and K.-H. Niggl. *Ranking primitive recursions: The low Grzegorczyk classes revisited.* SIAM J. comput. 29, No 2 (2000), 401-415.
7. S. J. Bellantoni and K.-H. Niggl and H. Schwichtenberg. *Higher type recursion, ramification and polynomial time.* Ann. Pure Appl. Logic 104 (2000), 17-30.
8. A. Cobham. *The intrinsic computational difficulty of functions.* Logic Methodology Philos. Sci., Proc. 1964 internat. Congr. 24-30 (1965).
9. R. Fagin. *Generalized first-order spectra and polynomial-time recognizable sets.* Complexity of Computation, SIAM-AMS Proceedings, 1974, Vol. 7, R. M. Karp , ed., 43-73.
10. A. Goerdt. *Characterizing complexity classes by higher type primitive recursive definitions.* Theor. Comput. Sci. 100, No.1 (1992), 45-66.
11. A. Goerdt and H. Seidl. *Characterizing complexity classes by higher type primitive recursive definitions, Part II.* Aspects and prospects of theoretical computer science (Smolenice, 1990), 148-158, LNCS 464, Springer, Berlin, 1990.
12. N.D. Jones. *Computability and complexity from a programming perspective.* MIT Press, Cambridge, MA, 1997.
13. N.D. Jones. *LOGSPACE and PTIME characterized by programming languages.* Theoretical Computer Science 228 (1999), 151-174.
14. N.D. Jones. *The expressive power of higher-order types or, life without CONS.* J. Functional Programming 11 (2001), 55-94.
15. L. Kristiansen. *Neat function algebraic characterizations of LOGSPACE and LINSPACE.* Computational Complexity (accepted).
16. L. Kristiansen and K-H. Niggl. *On the computational complexity of imperative programming languages.* Theoretical Computer Science 318 (2004), 139-161.

17. L. Kristiansen and P.J. Voda. *Complexity classes and fragments of C.* Information Processing Letters 88 (2003) 213-218.
18. L. Kristiansen and P.J. Voda. *The surprising power of restricted programs and Gödel's functionals.* In M. Baaz and J.A. Makowsky (Eds.): Computer Science Logic, LNCS 2803, pp. 345-358, Springer, 2003.
19. M. Kutylowski. *Small Grzegorczyk classes.* J. London Math. Soc. (2) 36 (1987), 193-210
20. D. Leivant. *Intrinsic theories and computational complexity.* In "Logical and Computational Complexity. Selected Papers. Logic and Computational Complexity, International Workshop LCC '94", Leivant (ed.), 177-194, Springer, 1995.
21. H. Lewis and C.H. Papadimitriou *Elements of the theory of computation.* Prentice-Hall, 1998.
22. K-.H. Niggl. *Control Structures in Programs and Computational Complexity.* To appear in Annals of Pure and Applied Logic.
23. P. Odifreddi. *Classical Recursion Theory, Vol II.* North-Holland, 1999.
24. R.W. Ritchie. *Classes of predictably computable functions.* Trans. Am. Math. Soc. 106, 139-173 (1963).
25. H.E. Rose *Subrecursion. Functions and hierarchies.* Clarendon Press, Oxford, 1984.
26. H. Schwichtenberg. *Complexity of normalization in the pure typed λ-calculus.* In A.S. Troelstra and D. Van Dalen (Eds.): The L.E.J. Brouwer Centenary Symposium, pp. 453-457, North-Holland, 1982.
27. H. Schwichtenberg and S.J. Bellantoni. *Feasible computation with higher types.* Available from Schwichtenberg's home page.
28. H. Simmons. *The realm of primitive recursion.* Arch. Math. Logic 27 (1988), 177-188.
29. H. Simmons. *Derivation and computation. Taking the Curry-Howard correspondence seriously.* Cambridge Tracts in Theoretical Computer Science. 51. (2000).
30. M.H.B. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism.* (Notes. Soon available as textbook.)