Complexity Classes and Higher Types

by Lars Kristiansen and Paul J. Voda

Abstract. We introduce an imperative programming language equipped with variables of higher types. Fragments of this language characterize complexity classes (including the small and important classes LOGSPACE, LINSPACE, PSPACE, P, EXP). Furthermore, we show how the same complexity classes can be characterized by fragments of Gödel's system T. All our characterizations can be dubbed implicit since they are purely syntactical with no references to explicit resource bounds.
Complexity Classes and Higher Types

Lars Kristiansen\textsuperscript{1,2} and Paul J. Voda\textsuperscript{3}

\textsuperscript{1} Oslo University College, Faculty of Engineering
Cort Adelers gate 30, N-0254 Oslo, Norway
\texttt{larskri@iu.hio.no} WWW home page: \url{http://www.iu.hio.no/~larskri}

\textsuperscript{2} Department of Mathematics, University of Oslo
Postboks 1053, Blindern, N-0316 Oslo, Norway

\textsuperscript{3} Institute of Informatics, Comenius University Bratislava
Mlynski dolina, 842 15 Bratislava, Slovakia
\texttt{voda@fsph.uniba.sk} WWW home page: \url{http://www.fsph.uniba.sk/~voda}

1 Introduction

For $i \in \mathbb{N}$, TIME $2^{n_i}$ (SPACE $2^{n_i}$) is the set of problem decidable by a Turing machine working in time (space) $2^{n_i}$ for some polynomial $p$; TIME $2^{n_i}$ (SPACE $2^{n_i}$) is the set of problem decidable by a Turing machine working in time (space) $2^{k|x|}$ for some $k \in \mathbb{N}$. Let STPOL\textsuperscript{0} = LOGSPACE, STPOL\textsuperscript{2i+2} = SPACE $2^{n_i}$ and STPOL\textsuperscript{2i+1} = TIME $2^{n_i}$. Let STLIN\textsuperscript{2i} = SPACE $2^{n_i}$ and STPOL\textsuperscript{2i+1} = TIME $2^{n_i+1}$. Then we have STPOL\textsuperscript{i} $\subseteq$ STPOL\textsuperscript{i+1} (STLIN\textsuperscript{i} $\subseteq$ STLIN\textsuperscript{i+1}) for each $i \in \mathbb{N}$, and $\bigcup_{i \in \mathbb{N}}$ STPOL\textsuperscript{i} ($\bigcup_{i \in \mathbb{N}}$ STLIN\textsuperscript{i}) is an alternating time-space hierarchy. (Note that we find familiar complexity like classes LOGSPACE, P, PSPACE, LINSpace etc. near the bottoms of the two hierarchies.) It is well known, and quite obvious, that STPOL\textsuperscript{i} $\subset$ STPOL\textsuperscript{i+2} (STLIN\textsuperscript{i} $\subset$ STLIN\textsuperscript{i+2}) holds for any $i \in \mathbb{N}$. Still, the problem STPOL\textsuperscript{n} = STPOL\textsuperscript{n+1} (STLIN\textsuperscript{n} = STLIN\textsuperscript{n+1}) is open for any fixed $n \in \mathbb{N}$.

In particular, the following problems are open: Does P equal LOGSPACE? Does P equal PSPACE? Does EXP equal LINSpace? Does EXP equal EXPSPACE? Complexity theorists in general, and the authors in particular, expect the answer to all these questions to be no, and it is a bit of a mystery that it should be so hard to prove that this indeed is the case. A further study of the two alternating time-space 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 STLIN\textsuperscript{2n} $\neq$ STLIN\textsuperscript{2n+1} it follows that P $\neq$ LOGSPACE and EXP $\neq$ LINSpace).

In this paper we give several characterizations of the two alternating space-time hierarchies. All our characterizations can be dubbed implicit since they are purely syntactical with no references to explicit resource bounds.

In Section 3 and 4 we introduce an imperative programming language equipped with variables of higher types. Fragments of this language capture the complexity classes in the alternating space-time hierarchies. (A related characterization of the small complexity classes near the bottoms of the hierarchies, that is the complexity classes LOGSPACE, P, PSPACE, LINSpace, EXP and EXPSPACE, can be found in Kristiansen and Voda [10].)
In Section 5 we introduce the calculus $T^\prec$. Roughly speaking, $T^\prec$ is Gödel’s $T$ where the successor function is removed. To be more precise, $T^\prec$ is the standard typed $\lambda$-calculus extended with a recursion $R_\tau$ for each type $\tau$, and the constant 1 of type $0$. A $T^\prec$-term $M$ has rank $n$ if we have $dg(\sigma) \leq n$ for every recursion $R_\sigma$ in $M$. (Here $dg(\sigma)$ denote the degree of the type $\sigma$.) Let $\mathcal{F}^n$ denote the set of problems that can be defined by a $T^\prec$-term of rank $n$. We prove that the hierarchy $\bigcup_{i \in \mathbb{N}} \mathcal{F}^i$ matches, level by level, the alternating space-time hierarchy $\bigcup_{i \in \mathbb{N}} \text{STLIN}^i$. Note that the classes in the hierarchy $\bigcup_{i \in \mathbb{N}} \mathcal{F}^i$ are defined in a uniform manner, whereas the classes in $\bigcup_{i \in \mathbb{N}} \text{STLIN}^i$ are not. The uniformity might yield a basis for finding a particular $n$ such that $\mathcal{F}^n \neq \mathcal{F}^{n+1}$.

The theorems and proofs in Section 5 can be modified to yield a similar uniform characterization of the hierarchy $\bigcup_{i \in \mathbb{N}} \text{STPOL}^i$.

Section 5 is inspired by the somewhat sketchy conference paper “Characterizing complexity classes by higher type primitive recursive definitions, Part II” by Goedt and Seidl [6] (and partially also by Goerdt [5]). Goerd and Seidel use finite model theory to characterize the hierarchy $\bigcup_{i \in \mathbb{N}} \text{STPOL}^i$. The inclusion $\mathcal{F}^n \subseteq \text{STLIN}^n$ should follow without too much ado from the theorems in [6]. Hopefully will our proofs in Section 5 be more enlightening and transparent than the proofs in [6] and [5]. The inclusion $\text{STLIN}^n \subseteq \mathcal{F}^n$ does not follow from Goerdt and Seidl’s work.

So-called ramification techniques (e.g. Simmons [18], Leivant [12], Beckmann and Weiermann [4]) restrict higher type recursion to the Kalmar elementary level, that is, to a complexity theoretic level. By using so-called linearity constraints in conjunction with ramification techniques, higher type recursion can be restricted further down to the “polynomial” level (e.g., Bellantoni and Schwichtenberg [16]). Ramification techniques in general are not very intuitive and tend to be intricate and opaque. We stress that no such techniques appear in this paper. We operate inside a neat and clean theoretic framework obtained by simply removing successor-like functions from a standard computability-theoretic framework. Jones [8] [9] uses the same technique. So do the authors in [10].

Parts of this paper (essentially Section 3) is published in the proceedings from CSL’03 [11].

2 Preliminaries

2.1 Numbers, Arrays and Types

**Definition.** $0$ is a type; $\sigma \rightarrow \tau$ is a type if $\sigma$ and $\tau$ are types. We say a type $\sigma$ has degree $n$ when $dg(\sigma) = n$ where $dg(0) = 0$ and $dg(\sigma \rightarrow \tau) = \max (dg(\sigma) + 1, dg(\tau))$. We define the cardinality of type $\sigma$ at base $b$, in symbols $|\sigma|_b$, by recursion on the build-up of $\sigma$: $|0|_b = b$ and $|\sigma \rightarrow \tau|_b = |\tau|_b^{\sigma|}$. We use standard conventions and interpret $\sigma \rightarrow \sigma' \rightarrow \sigma''$ by associating parentheses to the right, i.e. as $\sigma \rightarrow (\sigma' \rightarrow \sigma'')$; further, $\sigma, \sigma' \rightarrow \sigma''$ is alternative notation for $\sigma \rightarrow \sigma' \rightarrow \sigma''$. Let $2_0^x \equiv x$ and $2_{n+1}^x \equiv 2^{2^n x}$.
Lemma 1. (i) For any polynomial \( p \) and \( n > 0 \) there exists a type \( \sigma \) of degree \( n \) such that \( |\sigma|_{\max(x,1)} > 2_n^p(x) \). (ii) For every type \( \sigma \) of degree \( n \) there exists a polynomial \( p \) such that \( 2_n^p(x) > |\sigma|_x \).

Proof. (i) We prove (i) by induction on \( n \). Assume \( n = 1 \). Let the types \( \xi_0, \xi_1, \ldots \) be defined by \( \xi_0 = 0 \) and \( \xi_{j+1} = 0 \to \xi_j \). We have \( \deg(\xi_j) = 1 \) and \( |\xi_j| = x(2^j) \) for every \( j > 0 \) (*). (This is easily proved by induction on \( j \).) For an arbitrary polynomial \( p \) we have \( |\xi_k|_{\max(x,2)} > 2^p(x) \) for a sufficiently large \( k \). Thus, (i) holds when \( n = 1 \). We turn to the induction step. Assume that \( |\sigma|_{\max(x,1)} > 2_n^p(x) \) where \( \deg(\sigma) = n \). Then, we have

\[
2_{n+1}^p(x) = 2^{|\sigma|_{\max(x,1)}+1} \leq (|\sigma|_{\max(x,1)}+1)^n = |\sigma| \to 0_{\max(x,1)}+1
\]

and \( \deg(\sigma \to 0) = \deg(\sigma) + 1 \). (ii) We use induction on the build-up of \( \sigma \). The case \( \sigma = 0 \) is trivial. Assume \( \sigma = \tau \to \tau \). Then we have \( \deg(\rho) \leq n-1 \) and \( \deg(\tau) \leq n \), and the induction hypothesis yields polynomials \( p \) and \( q \) such that \( 2_n^p(x) > |\rho|x \) and \( 2_n^p(x) > |\tau|x \). We have

\[
|\sigma|_x = |\tau|_x^{|\rho|} < (2_n^p(x)|\tau|_x^{|\rho|} = 2^{|\tau|_x^{|\rho|}|\tau|_x^{|\rho|}} \leq 2_n^p(x) + q(x)
\]

and hence (ii) holds when \( p = r + q \). \( \square \)

Definition. The natural number \( a \) is a **number of type \( \sigma \) at base \( b \)**, in symbols \( a: \sigma_b \), iff \( a < |\sigma|_b \). Let \( a:(\sigma \to \tau)_b \). Then \( a \) 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 \( v_i : \tau_b \) for \( i \in \{0, \ldots, k\} \). 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 \( \sigma \) at base \( b \).) The notation \( a[i_1, \ldots, i_n]_b \), where \( n \geq 1 \), abbreviates \((a[i_1]_b)[i_2]_b \ldots [i_n]_b \), and we will call \( a[i_1, \ldots, i_n]_b \) a **sub-digit of \( a \)**. (Thus, every digit is a sub-digit, every digit of a digit is a sub-digit, and so on.) Further, let

\[
a[i_1, \ldots, i_{n+1}]_b := w]_b \overset{def}{=} a[i_1, \ldots, i_n]_b[i_{n+1} := w]_b
\]

for \( n \geq 1 \). Thus, \( a[i_1, \ldots, i_n]_b := w]_b \) is the number which is the result of setting the sub-digit \( a[i_1, \ldots, i_n]_b \) in \( a \) to \( w \). Occasionally we will suppress the base \( b \) in the notation, and we will also just talk about numbers of type \( \sigma \). Occasionally we will call a number of type \( \sigma \to \tau \) an **array**.

It might be helpful for readers familiar with programming languages to view the number \( a:(\sigma \to \tau)_b \) as an **array** with index set \( \sigma_b \) where each position \( a[i] \) in the array \( a \) (for \( i \in \sigma_b \)) is an element in the set \( \tau_b \). Equivalently, we can also view the number \( a:(\sigma \to \tau)_b \) as a **finite function**. \( \square \)
2.2 The Ritchie Hierarchy

Definition. Whenever we talk about the binary representation of a natural number \( z \), we will always mean the shortest possible binary representation, i.e. the one without any unnecessary leading zeroes.

We use standard Turing Machines with a read-only input tape, write-only output tape and one-way work tapes. Through the paper, the input to a Turing machine will always be one natural number \( x \), or several natural numbers \( \bar{x} \), given in binary representation; if the Turing machine computes a number-theoretic function, it should also give its output in the binary representation. We will use \( x (\bar{x}) \) to denote the input, and in time and space bounds \( x (\bar{x}) \) will denote a natural number(s) and not the length of any representation.

Let \( M \) be a Turing machine computing a number-theoretic function \( f (\bar{x}) \). We say that \( M \) works in space \( g(\bar{x}) \) if the number of cells visited on \( M \)'s work tape during the computation of \( f (\bar{x}) \) is bounded by \( g(\bar{x}) \). A Turing machine works in linear space if there exists a fixed number \( k \) such that the number of tape cells visited under a computation on the input \( w \) is bounded by \( k|w| \) where \( w \) denotes the length of the input.

We define the class \( \mathcal{R}^n \) of number-theoretic functions by

\[
\begin{align*}
&- f \in \mathcal{R}^0 \text{ iff } f \text{ is computable by a Turing machine working in linear space.} \\
&- f \in \mathcal{R}^{n+1} \text{ iff } f (\bar{x}) \text{ is computable by a Turing machine working in space } g(\bar{x}) \\
&\quad \text{ for some } g \in \mathcal{R}^n.
\end{align*}
\]

We define the Ritchie hierarchy \( \mathcal{R} \) by \( \mathcal{R} = \bigcup_{n\in \mathbb{N}} \mathcal{R}^n \).

Lemma 2. (i) For every function \( f \in \mathcal{R}^i \) there exists a polynomial \( p \) such that \( 2_i^p(\bar{x}) > f (\bar{x}) \). (In particular, any function in \( \mathcal{R}^0 \) is bounded by a polynomial.) (ii) For every polynomial \( p \) there exists a function \( f \in \mathcal{R}_{i+1} \) such that \( f (\bar{x}) > 2_{i+1}^p(\bar{x}) \). (In particular, any polynomial is bounded by a function in \( \mathcal{R}^0 \)).

Proof. (i) We prove (i) by induction on \( i \). It is easy to verify that for any \( f \in \mathcal{R}^0 \) there exists a polynomial \( p \) such that \( f (\bar{x}) < p(\bar{x}) = 2_0^p(\bar{x}) \). We skip the details. Assume \( f \in \mathcal{R}^{i+1} \). Then there exist a Turing machine \( M \) and \( g \in \mathcal{R}^i \) such that \( M \) computes \( f \) in space \( g(\bar{x}) \). The induction hypothesis yields a polynomial \( q(\bar{x}) \) such that \( M \) computes \( f \) in space \( 2_i^q(\bar{x}) \), and thus, the number of bits required to represent the value \( f (\bar{x}) \) is bounded by \( 2_i^q(\bar{x}) \). It follows that \( 2_{i+1}^q(\bar{x}) > f (\bar{x}) \) for some polynomial \( p \). (ii) We prove (ii) by induction on \( i \). We leave the case \( i = 0 \) to the reader, and focus on the induction step: Let \( p \) be an arbitrary polynomial. The induction hypothesis yields a function \( h \in \mathcal{R}^i \) such that \( h (\bar{x}) > 2_i^p(\bar{x}) \). Let \( f (\bar{x}) = 2^h(\bar{x}) \). Obviously we have \( f (\bar{x}) > 2_i^{h(\bar{x})} \), and we conclude the proof by argue that \( f \in \mathcal{R}^{i+1} \). A Turing machine \( M \) can compute \( f \) by the following procedure: First \( M \) computes the number \( h (\bar{x}) \) (this can obviously be done in space \( g_0 (\bar{x}) \) for some \( g_0 \in \mathcal{R}^i \)); then \( M \) computes \( f (\bar{x}) \) by writing down the digit 1 followed by \( h (\bar{x}) \) copies of the digit 0. Let \( g(\bar{x}) = g_0 (\bar{x}) + h (\bar{x}) + k \) where \( k \) is a sufficiently large fixed number. Then \( g \in \mathcal{R}^i \) and \( M \) works in space \( g \). Hence, we
have \( f \in \mathcal{R}^{i+1} \) since \( f \) can be computed by a Turing machine working in space \( g \) for some \( g \in \mathcal{R}^{i} \).

The hierarchy \( \mathcal{R} \) was introduced in Ritchie [14]. Historically, this must be one of the first papers where classes of functions or predicates are defined by putting resource bounds on Turing machine. Note that \( \mathcal{R}^{0} = \mathcal{E}^{2} \) where \( \mathcal{E}^{2} \) denotes the second Grzegorczyk class; \( \mathcal{R} = \mathcal{E}^{3} \) where \( \mathcal{E}^{3} \) denotes the third Grzegorczyk class also known as the class of Kalmár elementary functions.

### 2.3 Complexity classes

**Definition.** A problem is a set of natural numbers, i.e. a subset of \( \mathbb{N} \). If \( \mathcal{F} \) is a class of number-theoretic function, we use the standard notation \( \mathcal{F}_{n} \) to denote the set of problems which characteristic functions belongs to \( \mathcal{F} \), that is, \( \mathcal{F}_{n} = \{ A \mid \exists f \in \mathcal{F} \mid x \in A \iff f(x) = 0 \} \). We will use \( |x| \) to denote the length of the (shortest possible) binary representation of the natural number \( x \). (So \( |0| = 1, |1| = 1, |2| = 2, |3| = 2, |4| = 3, \ldots \).) For \( i \in \{0, \ldots, |x| - 1\} \), we let \( (x)_{i} \) denote the \( i \)th bit in the binary representation of \( x \). (Let \( (x)_{0} \) be the least and \( (x)_{|x|-1} \) the most significant bit.)

A Turing machine \( M \) decides a problem \( A \) on input \( x \in \mathbb{N} \) halts in a distinguished accept state if \( x \in A \), and in a distinguished reject state if \( x \not\in A \). The input \( x \in \mathbb{N} \) should be represented binary on Turing machine’s input tape. We view the standard complexity classes \( \text{LOGSPACE}, \text{P}, \text{PSPACE}, \text{LINSPACE}, \text{EXP} \) etc. as classes of problems, that is, as sets of subsets of \( \mathbb{N} \). So, \( \text{LOGSPACE} \) is the class of problems decidable by a Turing machine working in space \( k(\log_{2} |x| + 1) \) for some \( k \in \mathbb{N} \), \( \text{LINSPACE} \) is the class of problems decidable by a Turing machine working in space \( k|x| \) for some \( k \in \mathbb{N} \), etc. For \( i \in \mathbb{N} \), we define \( \text{TIME} 2^{i^{2x}} \) (space \( 2^{i^{2x}} \)) to be the set of problem decidable by a Turing machine working in time (space) \( 2^{i^{2x}} \) for some polynomial \( p \); we define \( \text{TIME} 2^{i^{2x}} \) (space \( 2^{i^{2x}} \)) to be the set of problem decidable by a Turing machine working in time (space) \( 2^{i^{2x}} \) for some polynomial \( p \). Assuming the nomenclature of Odifreddi [13] we have \( \text{TIME} 2^{0^{2x}} = \text{P}, \text{SPACE} 2^{0^{2x}} = \text{PSPACE}, \text{TIME} 2^{0^{2x}} = \text{POLYEXP}, \text{SPACE} 2^{0^{2x}} = \text{POLYEXPSPACE}, \text{SPACE} 2^{0^{2x}} = \text{LINS Shepard} \), \( \text{TIME} 2^{0^{2x}} \) = \( \text{EXP} \) and \( \text{SPACE} 2^{0^{2x}} = \text{EXPSPACE} \).

The following simple lemma is pivotal and will be used frequently, occasionally we will not explicitly refer to the lemma.

**Lemma 3.** (i) For any polynomial \( p \) there exists \( k \in \mathbb{N} \) such that \( p(x) < 2^{k|x|} \). (ii) For any \( k \in \mathbb{N} \) there exists polynomial \( p \) such that \( 2^{k|x|} < p(x) \). (iii) For any \( k \in \mathbb{N} \) there exists \( k' \in \mathbb{N} \) such that \( k|x| < k' \log_{2}(x + 2) \). (iv) For any \( k' \in \mathbb{N} \) there exists \( k \in \mathbb{N} \) such that \( k' \log_{2}(x + 2) < k|x| \).

**Proof.** First we note that \( x < 2|x| < 2(x + 1) \). Let \( p(x) \) be any polynomial, and let \( k_{0}, k_{1} \in \mathbb{N} \) be such that \( p(x) < x^{k_{0}} + k_{1} \). We have

\[
p(x) < x^{k_{0}} + k_{1} \leq (2^{k_{1}})^{k_{0}} + k_{1} = (2^{k_{1}})^{k_{0}} \leq 2^{(k_{0}+k_{1})|x|}.
\]
Thus, let $k = k_0 + k_1$, and we see that (i) holds. Further, (ii) holds since we for any $k \in \mathbb{N}$ have $2^k|x| = (2|x|)^k \preceq (2(x + 1))^k$. It is obvious that (iii) and (iv) holds.

The complexity classes defined in this this subsection contain problems whereas the classes in the Ritchie hierarchy contain number-theoretic functions. The next lemma settles the relationship between the Ritchie hierarchy and the complexity classes.

**Lemma 4.** $\mathcal{R}_i^k = \text{SPACE } 2_i^{1^n}$ for $i \in \mathbb{N}$.

**Proof.** First we note that we have $\mathcal{R}_i^k = \text{SPACE } 2_i^{1^n}$ straightaway from the definitions. Assume $A \in \mathcal{R}_{i+1}^k$. Then the problem $A$ can be decided by a Turing machine working in space $g$ where $g \in \mathcal{R}_i^k$. By Lemma 2 there exists a polynomial $p$ such that $g(x) < 2_i^{p(x)}$. By Lemma 3 there exists a $k \in \mathbb{N}$ such that $2_i^{p(x)} < 2_i^{p(x)}$. Hence $A \in \text{SPACE } 2_i^{1^n}$. Assume $A \in \text{SPACE } 2_i^{1^n}$. Thus, there exists $k \in \mathbb{N}$ such that $A$ is decided by a Turing machine working in space $2_i^{1^n}$. By Lemma 3 there exists a polynomial $p$ such that $2_i^{p(x)} < 2_i^{p(x)}$, and by Lemma 2 there exists $g \in \mathcal{R}_i^k$ such that $2_i^{p(x)} < g(x)$. Thus $A \in \mathcal{R}_i^{1^n}$. $\Box$

3 The Ritchie Hierarchy and While-programs

We will now define an imperative programming language. An informal explanation of the language follows the definition. It might be a good idea to read the definition and the explanation in parallel.

**Definition.** First we define the syntax of the programming language. The language has an infinite supply of program variables $x_0^0, x_1^1, x_2^2, \ldots$ for any type $\sigma$. We will use verbatim Latin letters, uppercase or lowercase, with or without subscripts and superscripts, to denote program variables. Any variable of type $\sigma$ is a term of type $\sigma$; $t[x]$ is an an term of type $\tau$ if $X$ is a variable of type $\sigma$ and $t$ is a term of type $\tau \rightarrow \sigma$. We use $a[i_1, \ldots, i_n]$ to abbreviate $a[i_1][i_2] \ldots [i_n]$. (Note that any term is a variable or has the form $a[i_1, \ldots, i_n]$ where $a, i_1, \ldots, i_n$ are variables.) The primitive instruction $t$ is a program in $\mathcal{X}$ if $t$ is a term of type $\sigma$ such that every variable in $t$ occurs in the variable list $\mathcal{X}$; the while-loop $\text{while } \{P\}$ is a program in $\mathcal{X}$ if $P$ is a program in $\mathcal{X}$, and $t$ is a term of type $\sigma$ such that every variable in $t$ occurs in the variable list $\mathcal{X}$; the sequence $P; Q$ is a program in $\mathcal{X}$ if $P$ and $Q$ are programs in $\mathcal{X}$.

We will now define the semantics of the programming language. Let $P$ be a program in $\mathcal{X} = x_1, \ldots, x_n$. The meaning of $P$ is a $(2n + 1)$-ary input-output relation $x_1, \ldots, x_n \{P\} y_1, \ldots, y_n$ over the natural numbers. We say that the arguments $x_i$ and $y_i$ (for $i = 1, \ldots, n$) are assigned to the program variable $x_i$ in the list $\mathcal{X} = x_1, \ldots, x_n$. We define the relation by recursion over the syntactical build-up of $P$:
The relation \( \mathcal{E} \{ Q ; R \} \) holds iff there exists \( \mathcal{E} \) such that \( \mathcal{E} \{ Q \} \) and \( \mathcal{E} \{ R \} \) hold.

The relation \( \mathcal{E} \{ \text{while} \{ Q \} \} \) holds iff there exists a sequence \( \mathcal{E}^{1}, \ldots, \mathcal{E}^{k} \) such that

- \( \mathcal{E} = \mathcal{E}^{1} \) and \( \mathcal{E}^{1} = \mathcal{E}^{k} \)
- \( \mathcal{E}^{1} \{ Q \} \mathcal{E}^{i+1} \) holds for \( i \in \{ 1, \ldots, k - 1 \} \).
- \( t^i \neq 0 \) for \( i \in \{ 1, \ldots, k - 1 \} \) and \( t^k = 0 \), where \( t^i \) is the the interpretation.

The relation \( \mathcal{E} \{ z, \mathcal{G} \{ Z \} \} \) holds iff \( z \) is assigned to the type 0 variable \( Z \) and \( z' = z \oplus b \).

The relation \( \mathcal{E}, a, \mathcal{G} \{ a [i_1, \ldots, i_m] + \} \{ b \mathcal{E}, a', \mathcal{G} \} \) holds iff the number \( a \) is assigned to \( a \), the numbers \( i_1, \ldots, i_m \) are respectively assigned to \( i_1, \ldots, i_m \) and

\[
a' = a[i_1, \ldots, i_m] := (a[i_1, \ldots, i_m] \oplus b 1) \).
\]

This completes the definition of the programming language.

A program \( P \) in the variables \( x_1, \ldots, x_k \) where \( x_1, \ldots, x_n \) are of type 0, computes the number-theoretic function \( f(x_1, \ldots, x_n) \) when

\[
\forall x_{n+1}, \ldots, x_k \exists y_1, \ldots, y_{k-1} [ \; x_1, \ldots, x_n, x_{n+1}, \ldots, x_k \{ P \} b y_1, \ldots, y_{k-1}, z \; ]
\]

iff \( f(x_1, \ldots, x_n) = z \)

where \( b = \max(x_1, \ldots, x_n, 1) + 1 \)

(Note that the output variable \( Z \) can be of any type.) A program decides a number-theoretic relation if it computes the characteristic function of the relation. A program \( \mathcal{E} \) of type \( \sigma \) has degree \( n \) when \( \sigma \) has degree \( n \). A program \( \mathcal{E} \) of type \( \sigma \) has degree \( n \) when every variable in \( \mathcal{E} \) has degree \( \leq n \).

Let \( \mathcal{P}^{n} \) denote the set of number-theoretic functions computed by the programs of degree \( n \). The program hierarchy \( \mathcal{P} \) is defined by \( \mathcal{P} = \bigcup_{n \in \mathbb{N}} \mathcal{P}^{n} \).

Finally, we need some notation to develop, specify and reason about programs. Let \( \mathcal{E} \) be a list of variables, and let \( P \) be a program in \( x_1, \ldots, x_n \). We will use \( \mathcal{E} \{ P \} \) to abbreviate \( \forall \exists \exists \mathcal{E} \mathcal{U} \mathcal{E} \{ P \} \). We will develop programs in an informal mnemonic notation. The notation should need no further explication for readers familiar with standard programming languages. We will always assume that the local macro variables in the macros are suitably renamed such that we avoid name conflicts. In the macros we will use uppercase Latin letters \( \mathcal{E}, Y, Z, \) etc. to denote variables of type 0, and lowercase Latin letters \( u, i, j, \) etc. to denote variables of higher types.

In the following we give an explication of the programming language. The execution of program will take place in a certain base \( b \) given by \( b = \max(\mathcal{E}, 1) + 1 \) where \( \mathcal{E} \in \mathbb{N} \) are the inputs. Program variables of type \( \sigma \) store natural numbers in the set \( \{0, 1, \ldots, |\sigma| - 1\} \). The only primitive instruction in the language has either the form (i) \( a [x_1, \ldots, x_n] \) where \( a, x_1, \ldots, x_n \) are variables such that \( a [x_1, \ldots, x_n] \) is a type 0 term, or the form (ii) \( Y + \) where \( Y \) is a type 0 variable.

In case (i) the sub-digit \( a [x_1, \ldots, x_n] \) in \( a \) is increased by 1 modulo \( b \), that is,
the sub-digit will be increased by 1, except in the case when the sub-digit equals \( b - 1 \), then it is set to 0. In case (ii) \( Y \) is increased by 1 modulo \( b \). The language has one control structure, the loop \( \text{while } t \{ \ldots \} \) where \( t \) is a term of the same form and type as to those allowed in the primitive instruction. This is a standard while-loop executing its body repeatedly as long as the value (sub-digit) \( t \) refers to is different from 0. The semicolon composing two programs has the standard meaning.

**Definition.** Let \( x \oplus_n y \equiv (x + y) \) (mod \( n \)) and \( x \ominus_n y \equiv (x - y) \) (mod \( n \)). Let \( sg(x) = 0 \) if \( x = 0 \); otherwise \( sg(x) = 1 \).

**Lemma 5.** Let \( X, Y \) and \( Z \) denote type 0 variables, and assume that \( x, y \) and \( z \) are assigned to respectively \( X, Y \) and \( Z \). Let \( b \geq 2 \) and assume \( x, y, z < b \). The following macros can be implemented as type 0 programs.

- \( x \{ X:=0 \}_b 0 \) (assignment of the constant 0)
- \( x \{ X:=1 \}_b 1 \) (assignment of the constant 1)
- \( x, y \{ X:=x \}_b x, x \) (assignment)
- \( z, x, y \{ Z:=x+Y \}_b x \oplus_b y, x, y \) (addition modulo \( b \))
- \( z, x, y \{ Z:=x-Y \}_b x \ominus_b y, x, y \) (subtraction modulo \( b \))
- \( x, y \{ X:=sg \}_b sg(y), y \) (converting numbers to boolean values)
- \( x, y \{ X:=\text{not} \}_b x', y \) where \( x' = 1 \) if \( y = 0 \); otherwise \( x' = 0 \) (logical not)
- \( z, x, y \{ Z:=\text{and} \}_b z', x, y \) where \( z' = 1 \) if \( x = y = 0 \); otherwise \( z' = 0 \) (logical and)

**Proof.** Let \( X:=0 \) \( \equiv \) \( while \ X \{+\} \) and let \( X:=1 \equiv X:=0; X+. \) Let \( \overline{X} \) denote the complement of \( x \) modulo \( b \), i.e. \( \overline{X} \) is the unique number such that \( x \oplus_b \overline{X} = 0 \). Let

\[
cc(X,Y,Z) \equiv Y:=0; Z:=0; while \ X \{ X++; Y++; Z+ \}.
\]

Then we have \( x, y, z \) \( cc(X,Y,Z) \) 0, \( \overline{X} \). Note that \( \overline{X} = x \). Thus, let

\[
Y:=\overline{X} \equiv cc(X,U,V); cc(U,X,Y).
\]

Further, let \( Z:=X+Y \equiv Z:=X; cc(Y,U,V) \); while \( U \{ Z++; Y++; U+ \}. \) By using the macros we have defined so far we can also easily define the macro \( Z:=X-Y \) since \( x \oplus_b y = \overline{\overline{x} \oplus_b y} \). Let \( X:=\text{not} \equiv X:=Y; X:=1 \); while \( Z \{ X:=0; Z+ \} \) and let \( X:=sg \equiv U:=\text{not} \); \( X:=\text{not} \). We see that also \( Z:=\text{or} \) \( Y \) can be implemented since \( sg(sg(x) \oplus_b sg(y)) = 0 \) if \( x = y = 0 \) and 1 otherwise. Use \( X:=\text{not} \) and \( Z:=X \) \( Y \) to implement \( Z:=X \) \( Y \).

\( \square \)

Lemma 5 tells that the type 0 fragment of the programming language is powerful. Indeed the fragment yields full Turing computability if the programs always are executed at a sufficiently high base, i.e. the base \( b \) is set to a sufficiently large number before the execution starts. We will prove that if the base \( b \) is set to \( \max(\overline{X}, 1) + 1 \) (where \( \overline{X} \) are the inputs) the type 0 fragment capture the complexity
class LINSPACE. Programs containing higher types can compute functions beyond LINSPACE. When we set the base \( b \) to \( \max(x,1) + 1 \), type 0 programs can do arithmetic modulo \( b \); in general, programs of type \( \sigma \) can do do arithmetic modulo \( \|\sigma|_b \).

**Lemma 6.** Let \( b \geq 2 \) and let \( c = |\sigma|_b \). Further, Let \( u, v \) and \( w \) be program variables of type \( \sigma \), and let \( x \) be a program variable of type 0. Assumed that the numbers \( u, v, w \) and \( x \) (where \( u, v, w < c \) and \( x < b \)) are assigned to respectively \( u, v, w \) and \( x \). The following macros can be implemented as type \( \sigma \) programs.

- \( u \{ u := 0 \}_b \) (assignment of 0 to a variable of type \( \sigma \))
- \( u \{ u = \_ \}_b \) (successor modulo \( |\sigma|_b \))
- \( u \{ u = \_ \}_b \) (predecessor modulo \( |\sigma|_b \))
- \( u, x \{ x := \text{sg}(u) \}_b, u, \text{sg}(x) \)
- \( u, v \{ u := v + w \}_b, v \oplus u, w, v, w \) (addition modulo \( |\sigma|_b \))
- \( u, v \{ u := v - w \}_b, v \ominus u, w, v, w \) (subtraction modulo \( |\sigma|_b \))

**Proof.** We define the three macros \( u_\_ \), \( u := 0 \) and \( X := \text{sg}(u) \) simultaneously by recursion over the build-up of the type \( \sigma \). By Lemma 5, we can define the macros when \( \sigma \equiv 0 \). Now, assume \( \sigma \equiv \pi \to \tau \) and that the macros are defined for the types \( \pi \) and \( \tau \). Then we define \( u := 0 \) by

\[
i := 0; X := 1 \text{ while } u \{ u[i] := 0; i = 0; \}
\]

(Explanation: \( u \) is a \(|\tau|_b\)-digit number in base \(|\tau|_b \). The macro sets each digit to 0.) We define \( X := \text{sg}(u) \) by

\[
i := 0; U := 1; X := 0 \text{ while } U \{ X := \text{sg}(u[i])_\pi; i := 0; \}
\]

(Explanation: \( u \) is a \(|\tau|_b\)-digit number in base \(|\tau|_b \). The macro sets the type \( \pi \) variable \( X \) to 0 if each digit in \( u \) is 0.) We define \( u_\_ \) by

\[
i := 0; j := 0; j := j + 1 \text{ while } U \{ j := j + 1; u[i] := 0; U := \text{sg}(u[i])_\tau; \}
\]

(Explanation: \( u \) is a \(|\tau|_b\)-digit number in base \(|\tau|_b \). The macro increases the digit \( u[0] \) by 1 modulo \(|\tau|_b \). If the digit turns into 0, the macro increases the digit \( u[1] \) by 1 modulo \(|\tau|_b \). If the digit turns into 0, the the digit \( u[2] \) is increased, and so on. Note, when the execution of the loop’s body starts, we have \( i \oplus 1 = j \) where \( d = |\tau|_b \). Given the macros \( u_\_ \), \( u := 0 \) and \( X := \text{sg}(u) \), it is easy to define \( u := v + w \) and \( u := v - w \). Implement \( u := v + w \) and \( u := v - w \) by computing complements, i.e. take advantage of the equation \( x \ominus y = \overline{\overline{x} \ominus \overline{y}} \) where \( \overline{\overline{\cdot}} \) denotes the \( c \)-complement of \( \cdot \), i.e. the unique number such that \( \cdot + c = c \).

**Theorem 1.** The program hierarchy and the Ritchie hierarchy match from level 1, i.e. \( P^{i+1} = R^{i+1} \) for any \( i \in \mathbb{N} \).
\textbf{Proof.} We first prove $P^{i+1} \subseteq R^{i+1}$. Assume $f \in P^{i+1}$. Then there exists a program $P$ of degree $i+1$ computing $f(\vec{x})$. By Lemma 1, there exists a polynomial $p$ such that no variable exceeds $2^{p(x)}$ during the execution. Let $M$ be a Turing machine which straightforwardly simulates the execution of the program $P$. Let us say there are $k$ variables in $P$. Then $M$ needs roughly $k2^p(x)$ tape cells to trace the contents of the variables during the execution. ($M$ represents numbers in binary notation.) Hence, there exists a polynomial $q$ such that $M$ runs in space $2^{q(x)}$. By Lemma 2 (ii) there exists $g \in R^i$ such that $M$ runs in space $g(\vec{x})$. This proves that $f \in R^{i+1}$.

We will now prove $R^n \subseteq P^n$. The proofs splits into two cases (1) $n = i + 2$, (2) $n = 1$. Case (1). Let $f \in R^{i+2}$. We can without loss of generality assume that $f$ is a unary function. By Lemma 2, there exist a polynomial $p$ and a Turing machine $M$ computing $f$ such that the number of cells visited on $M$’s tape in the computation of $f(x)$ is bounded by $2^{p(x)}$. We will, uniformly in the Turing machine $M$, construct a program computing $f$. Let $|\Sigma|$ denote the cardinality of $M$’s alphabet (including the blank symbol). If $\max(x, 1) < |\Sigma|$, the program computes $f(x)$ by a hand tailored algorithm, i.e. the program simply consults a finite table. If $\max(x, 1) \geq |\Sigma|$, the program computes $f(x)$ by simulating $M$.

We map each symbol in $M$’s alphabet to a unique value in the set $\{0, \ldots, |\Sigma|\}$. (This is possible since $\max(x, 1) \geq |\Sigma|$) By Lemma 1 there exists a type $\tau$ of degree $i + 1$ such that $|\tau|^{\max(x, 1)+1} > 2^{p(x)}$. Thus, the program can represent $M$’s tape by a variable $a$ of type $\tau \rightarrow 0$, and the scanning head by variable $x$ of type $\tau$. The scanning head can be moved one position to the right (left) by executing the program $x_{+\tau} \ (x_{-\tau})$, and the type 0-term $a[X]$ yields the scanned symbol. The degree of the program will equal the degree of $a$, i.e. $i + 2$. Thus, $f$ can be computed by a program degree $i + 2$. Hence, $f \in P^{i+2}$. Case (1). We skip this case.

\textbf{Definition.} A function $f$ is \emph{non increasing} when $f(\vec{x}) \leq \max(\vec{x}, 1)$.

The reason that the Ritchie hierarchy and the program hierarchy do not match at level 0, is simply that type 0 programs compute only non increasing functions. The next theorem says that with respect to non increasing functions the hierarchies also match at level 0.

\textbf{Theorem 2.} We have $f \in P^0$ iff $f \in R^0$ for every non increasing function $f$.

\textbf{Proof.} Assume $f \in P^0$. Let $P$ be a type 0 program computing $f(\vec{x})$. No variable in $P$ will exceed $\max(\vec{x}, 1)$ during the computation of $f(\vec{x})$. Let $M$ be a Turing machine which straightforwardly simulates $P$. Representing numbers binary $M$ needs $(\log \max(\vec{x}) + 1) + 1$ tape cells to trace the content of one variable during the simulation. Thus, it is easy to see that $f$ can be computed by a Turing Machine working in linear space, i.e. $f \in R^0$.

Assume that the Turing machine $M$ computes the function $f(\vec{x})$ in linear space. First we will argue that $M$ can be simulated by a type 0 program for all but finitely many inputs. The number of cells visited on $M$’s work tape during
the computation of $f(\vec{x})$, is strictly bounded by $k' \max(\vec{x})$ for some fixed $k' \in \mathbb{N}$. Assume there are $r$ symbols in $M$’s alphabet (including the blank symbol $B$). We view these symbols as digits, and in particular we view the blank symbol $B$ as a digit different from zero. A string of these symbols, which starts with $B$, and has length $k' \max(\vec{x})$, can be viewed as a natural number $t$ in base $r$. It now becomes clear how one natural $t$ number can represent one possible work tape configuration in the execution of $M$ on input $\vec{x}$: The $i + 1$’th digit in the base $r$ representation of $t$ is $k$ iff the symbol in the $i$’th tape cell is $k$. Moreover, for each such $t$ we have $t < r^{k' \max(\vec{x})}$. A simple arithmetical argument yields fixed $k, n \in \mathbb{N}$ such that $r^{k' \max(\vec{x})} \leq \max(\vec{x}, k)^n$. Hence it will be sufficient to use natural numbers in the range $0, \ldots, \max(\vec{x}, k)^n - 1$ to represent the work tape configurations in the execution of $M$ on input $\vec{x}$. Any number in this range can be represented by an $n$-tuple $(z_1, \ldots, z_n)$ where $z_i < \max(\vec{x}, k)$ (for $i = 1, \ldots, n$). Thus, we see that a fixed number of type 0 variables can contain enough information to represent the work tape if a program is executed at base $b$ where $b \geq \max(\vec{x}, k)$ It follows that we can construct a type 0 program $P$ computing the non increasing function $f$, i.e. $P$ such that

$$\exists t [ \vec{x}, \emptyset \{P\}_{\max(\vec{x})+1} \quad t, t] \quad \text{iff} \quad f(\vec{x}) = t.$$ 

If it is the case that $\max(\vec{x}, 1) + 1 \geq \max(\vec{x}, 1)$, then $P$ computes $f$ by simulating $M$. If $\max(\vec{x}, 1) + 1 < \max(\vec{x}, k)$, then $P$ computes $f$ by a hand tailored algorithm. (It will be the case that $\max(\vec{x}, 1) + 1 < \max(\vec{x}, k)$ for only finitely many values of $\vec{x}$. Thus the program can compute $f(\vec{x})$ by consulting a fixed table.)

The following lemma is needed in the succeeding section. We leave the proof to the reader.

**Lemma 7.** Let $k > 0$, and let $P$ be a program of degree $k$ computing the total function $f(\vec{x})$. There exists a type $\sigma$ of degree $k$ such that the number of steps $P$ uses to compute $f(\vec{x})$ is bounded by $|\sigma|_{\max(\vec{x})+1}$. (One step corresponds to the execution of one primitive instruction, i.e. an instruction of the form $t+.\)

### 4 Complexity Classes and For-programs

**Definition.** We extend the programming language given in Section 3. Syntax: for every type $\sigma$ the for-loop for $\sigma$ $\{P\}$ is a program in $\vec{x}$ if $P$ is a program in $\vec{x}$; the conditional if $t$ $\{P\}$ is a program in $\vec{x}$ if $P$ is a program in $\vec{x}$, and $t$ is a term of type 0 such that every variable in $t$ occurs in the variable list $\vec{x}$. Semantics:

- The relation $\vec{x}\{\text{for }\sigma \{Q\}\}_{\vec{y}\vec{y}}$ holds iff there exists a sequence $\vec{x}^1, \ldots, \vec{x}^k$ where $k = |\sigma|$ such that
  - $\vec{x}^i = \vec{z}^i$ and $\vec{y}^i = \vec{z}^i$
  - $\vec{x}^i\{Q\}_{\vec{z}^i+1}$ holds for $i \in \{1, \ldots, k - 1\}$.
- Let $n \in 0$ be the interpretation of the type 0 term $t$ under the assignment $\vec{x}$ to the program variables. The relation $\vec{x}\{\text{if } t \{Q\}\}_{\vec{y}\vec{y}}$ holds iff
  $$(n = 0 \land \vec{x} = \vec{y}) \lor (n \neq 0 \land \vec{x}\{Q\}_{\vec{y}\vec{y}}).$$
This completes the extension of the programming language.

The next definitions might be hard to digest. First, recall that a problem is nothing but a set of natural numbers. Informally, a program decides a problem \( A \) when the program answers the question “does the natural number \( x \) belong to the set \( A \)” correctly. The program should terminate on any input, and the answer will be found in a dedicated output variable of type \( 0 \). If the output variable contains \( 0 \) the answer is “yes”; otherwise, the answer is “no”. The input \( x \in \mathbb{N} \) should be held in a dedicated input variable when the program starts, and a program will either work in type 0 mode or in type 1 mode. A program working in type 0 mode, is executed in the base \( b = \max(x, 1) + 1 \) and has access to the input \( x \) in a variable of type \( 0 \). A program working in type 1 mode, is executed in the base \( b = |x| + 1 \) and has access to the bits in the binary representation of the input \( x \). The input variable holds an array \( a \) of type \( 0 \rightarrow 0 \), and \( a[i]_b \) yields the \( i \)'th bit in the binary representation of \( x \) (for \( i \in \{0, \ldots, |x| - 1\} \)). Let us turn to the formal definitions.

**Definition.** The program \( P \) in the variables \( X, Y, Z \) decides the problem \( A \) in type 0 mode when

\[
\forall \bar{y}, z \exists x', \bar{y}' [ x, \bar{y}, z \{P \} \ b \ x', \bar{y}', 0 ] \text{ iff } x \in A
\]

where \( b = \max(x, 1) + 1 \), and the variables \( X, Z \) have type \( 0 \). The program \( P \) in the variables \( X, Y, Z \) decides the problem \( A \) in type 1 mode when

\[
\forall \bar{y}, z \exists x', \bar{y}' [ a, \bar{y}, z \{P \} \ b \ x', \bar{y}', 0 ] \text{ iff } x \in A
\]

where \( b = |x| + 1; a[i]_b = (x)_i \) for \( i \in \{0, \ldots, |x| - 1\} \), i.e. the \( i \)'th bit in the binary representation of \( x \) should equal the \( i \)'th digit in \( a \) when \( a \) is interpreted as a \( b \)-digit number in base \( b \); the variables variables \( X, Z \) have respectively type \( 0 \rightarrow 0 \) and type \( 0 \). We will say that programs deciding problems the in type 0 (resp. 1) mode, receive the input as a type 0 (resp. 1) object and that they work in type 0 (resp. 1) mode.

We will now define fragments of our programming language. Each fragment will decide a set of problems, and in each case this set will equal a complexity class.

**Definition.** A while-program contains no for-loops and no conditional statements. A for-program contains no while-loops. A program has data degree \( n \) when every variable occurring in a program instruction on the form \( t^* \), has degree \( \leq n \). A for-loop has degree \( n \) where \( n = \deg(\sigma) \). A for-program has loop degree \( n \) when every loop in program has degree \( \leq n \).

A problem \( A \) belongs to the class \( \mathcal{W}_{\text{sen}}^{i,j} \) iff \( A \) can be decided in the type \( m \) mode by some while-program of data degree \( i \). A problem \( A \) belongs to the class \( \mathcal{L}_{\text{sen}}^{i,j} \) iff \( A \) can be decided in the type \( m \) mode by some for-program of loop degree \( i \) and data degree \( j \).
Note that a for-program has both a data degree and a loop degree; that a while-program has a data degree, but no loop degree. Further, note that the variable $x$ of degree $> n$ might occur in a program of data degree $n$, but $x$ cannot occur in an instruction on the form $t^*$. Such variables cannot be updated, they are read-only variables. Thus, the definition of classes $W^i_{\text{rvns}}$ and $L^i_{\text{rvns}}$ makes sense. A program can receive the input as a type 1 object, read the input and still have data degree 0.

**Lemma 8.** For any $i \in \mathbb{N}$ we have (i) $W^i_{\text{rvns}} = \text{SPACE } 2^i$ and (ii) $W^{i+1}_{\text{rvns}} = \text{SPACE } 2^i$. Besides, $W^0_{\text{rvns}} = \text{LOGSPACE}$.

**Proof.** It follows straightforwardly from Theorem 1 and Theorem 2 that $\mathcal{R}^i = P^i = W^i_{\text{rvns}}$. (The equality $P^i = W^i$ is obvious.) Now (i) follows from Lemma 4.

Recall that a problem belongs to $\text{SPACE } 2^i$, iff it can be decided by a Turing machine working in space $2^i$ for some fixed $k \in \mathbb{N}$. Hence, (i) together with Lemma 3 yield the following claim.

**(Claim I)**

1. $A \in W^i_{\text{rvns}}$ iff $A$ is decidable by a Turing machine working in space $2^i$ for some fixed polynomial $p$.
2. $A \in W^0_{\text{rvns}}$ iff $A$ is decidable by a Turing machine working in space $k \log_2 (x+2)$ for some fixed $k \in \mathbb{N}$.

The following claim does also hold.

**(Claim II)**

1. $A \in W^i_{\text{rvns}}$ iff $A$ is decidable by a Turing machine working in space $2^i$ for some fixed polynomial $p$.
2. $A \in W^0_{\text{rvns}}$ iff $A$ is decidable by a Turing machine working in space $k \log_2 (2^i+1)$ for some fixed $k \in \mathbb{N}$.

Note the symmetry between the two claims, and note that the bounds on the Turing machines in (Claim II) are given in the length of the input $|x|$, whereas the corresponding bounds in (Claim I) are given in the input $x$. The following argument elucidates the symmetry and explains how the proof of (Claim I), essentially given in Section 3, can be turned into a proof of (Claim II): A while-program of degree $i+1$ receiving the input $x$ as a type 1 object (in contrast to a type 0 object) will be executed in the base $b = |x| + 1 = \max(|x|, 1) + 1$ (in contrast to $b = \max(x, 1) + 1$). Hence, the program can be simulated by a Turing machine working in space $2^i$ for some polynomial $p$.

The other way around, a Turing machine working in space $2^i$ (in contrast to $2^i$), can for all but finitely many inputs be simulated by a while-program of degree $i+1$ executed in the base $b = |x| + 1 = \max(|x|, 1) + 1$ (in contrast to base $b = \max(x, 1) + 1$).

Clause (ii) and Clause (iii) of the theorem follow respectively from the first and second part of (Claim II). □
Definition. A program P in the variables \( x_1, \ldots, x_n, x_{n+1}, \ldots, x_k \) implements a program Q in the variables \( x_1, \ldots, x_n \) when

\[
x_1, \ldots, x_n \{ Q \} b x'_1, \ldots, x'_n \quad \text{iff} \quad \forall x_{n+1}, \ldots, x_k \exists x'_{n+1}, \ldots, x'_k \left[ x_1, \ldots, x_k \{ P \} b x'_1, \ldots, x'_k \right].
\]

for all \( b > 1 \) and all \( x_1, \ldots, x_n, x'_1, \ldots, x'_n \in \mathbb{N} \).

Lemma 9. (i) Every for-program of loop degree \( n \) data degree \( n \) can be implemented as a while-program of data degree \( n \). (ii) Every while-program of data degree \( n \) can be implemented as a for-program of loop degree \( n \) and data degree \( n \). (Hence, a problem belongs to \( \mathcal{L}^{n,n}_{\text{for}} \) iff it belongs \( \mathcal{W}^{n,n}_{\text{for}} \) for \( n \in \mathbb{N} \) and \( j \in \{0,1\} \).

Proof. (i) holds since the loop \( \text{for}_\tau \{ P \} \) can be implemented by

\[
P; \ X := x_0; \ X^+_\tau; \text{while } X \{ P; \ X^+_\tau \}
\]

where \( X \) is a fresh variable of type \( \sigma \). (The macros \( X := x_0 \) and \( X^+_\tau \) are defined in Lemma 6.) We prove (ii). Let P be a while-program of degree \( n \). By Lemma 1 there exists a fixed polynomial \( p \) such that \( 2^{n[x]} > \max(\sigma_1[x], \ldots, \sigma_k[x]) \) where \( \sigma_1, \ldots, \sigma_k \) are the types of the variables occurring in P. Thus, if a loop in P terminates at all, it will terminate before the body is executed \( k \times 2^{n[b]} \) times (where \( b \) is the base of the execution). Lemma 1 yields a type \( \tau \) of degree \( n \) such that \( |\tau|_b > k \times 2^{n[b]} \), and we can implement any one loop while \( t \{ P \} \) occurring in P by the program \( \text{for}_\tau \{ \text{ifn } t \{ P \} \} \).

Lemma 10. For all \( i \in \mathbb{N} \) we have (i) \( \mathcal{L}^{i,i+1}_{\text{for}} \subseteq \text{TIME } 2^{n[x]} \) and (ii) \( \mathcal{L}^{i,i+1}_{\text{for}} \subseteq \text{TIME } 2^{n[x]} \).

Proof. Let P be a for-program with loop degree \( i \) and data degree \( i+1 \) taking the input \( x \) as a type 1 object. To prove (i), it is sufficient argue that there exists a Turing machine \( M \) and a polynomial \( p \) such that \( M \) simulates P in time \( 2^{n[x]} \). The Turing machine \( M \) simulates P the natural way tracing the contents of the variables. There are two types of primitive operations in P: (1) instructions on the form \( t + \) and (2) checking if the body of a conditional \( \text{ifn } t \{ \ldots \} \) should be executed, that is, checking if the type \( 0 \) term \( t \) equals 0. Since P has loop degree \( i \) and takes the input as a type 1 object, there will be a polynomial \( p_i \) such that \( 2_{i}^{n[x]} \) puts an upper bound on the number of times a particular loop will execute its body. The loops in P are of course nested to a finite depth, and thus there will also exists a polynomial \( p_i \) such that \( 2_{i}^{n[x]} \) puts an upper bound on the number of primitive operations \( M \) needs to simulate. We will now argue that there also exists a polynomial \( q_i \) such that \( M \) can simulate each such primitive operation in time \( 2_{i}^{n[x]} \). It follows that \( M \) runs in time \( 2_{i}^{n[x]} \) for some polynomial \( p \).

The data degree of P is \( i+1 \) and the execution base \( b \) equals \( |x|+1 \). Thus there exists a polynomial \( q_0 \) such that \( 2_{i+1}^{n[x]} \) puts an upper bound on the numbers P
will store in a variable during the execution. Let us say there are $k$ variables in P. Then, representing numbers binary, $M$ needs roughly $k \times 2^{q_1[x]}$ tape cells to trace the contents of the variables. It is easy to see that there exists a polynomial $q_1$ such that one primitive operation can be carried out by $M$ in $q_1(k \times 2^{q_1[x]})$ steps. Thus there exists a polynomial $q$ such that any primitive operation can be simulated by $M$ in time $2^q[x]$. This completes the proof of (i). To see that (ii) holds, note that

- the execution base $b$ will equal $\max(x, 1) + 1$ for programs receiving the input $x$ as a type 0 object
- for any polynomial $p$ there exists a fixed number $k$ such that $p(x) \leq 2^k |x|$. Keeping these two facts in mind, the proof of (i) can easily be transformed into a proof of (ii).

Lemma 11. For all $i \in \mathbb{N}$ we have (i) TIME $2^{|x|} \subseteq L_{\text{for}}[i+1]$, and (ii) TIME $2^{|x|+1} \subseteq L_{\text{for}}[i+1]$.  

Proof. Let $M$ be a Turing machine running in time $2^{|x|}$ where $p$ is a primitive polynomial. To prove (i), it is sufficient to argue that $M$ can be simulated by a for-program of loop degree $i$ and data degree $i + 1$. By Lemma 1 we can pick a type $\sigma$ of degree $i$ such that $|\sigma|_{x+1} > 2^{|x|}$. Hence, when the execution base $b = |x| + 1$, the loop for $\sigma$ (\ldots) repeats its body more than $2^{|x|}$ times. Thus, using for $\times$-loops in place of the while-loops, we can simulate $M$ in the same manner as we simulate the Turing machine in the proof of Theorem 1. We use the operations $\times_\#$ and $\times_\sigma$ to simulate the movements of $M$'s scanning head. These operations can obviously be implemented as for $\sigma$-programs. We use a variable of type $\sigma \rightarrow \text{0}$, and hence, of degree $i + 1$, to represent the tape. No other variable in the program simulating $M$, will have higher degree. Thus, $M$ can be simulated by a program of loop degree $i$ and data degree $i + 1$. This completes the proof of (i). To see that (ii) also holds simply note that for every fixed $k \in \mathbb{N}$ exists a type $\sigma$ of degree $i$ such that $|\sigma|_{\max(x,1)+1} > 2^k |x|$, and that the execution base $b$ equals $\max(x, 1) + 1$ when the program receive the input as a type 0 object.

Theorem 3. For all $i \in \mathbb{N}$ we have (i) TIME $2^{|x|} \subseteq L_{\text{for}}[i+1]$, (ii) TIME $2^{|x|+1} = L_{\text{for}}[i+1]$, (iii) SPACE $2^{|x|} = L_{\text{for}}[i+1]$ and (iv) SPACE $2^{|x|+1} = L_{\text{for}}[i+1]$. Besides, we have (vi) LOGSPACE = $L_{\text{for}}[0]$, See also Table 1.

Proof. The theorem follows straightforwardly from the four preceding lemmas.

5 Fragments of Gödel's $T$

5.1 Preliminaries and definitions

We will now extend some of our definitions from Section 2 with product types.
Table 1. The table relates the equations in Theorem 3 to a fairly standard nomenclature used e.g. in Odifreddi [13].

<table>
<thead>
<tr>
<th>Degree of the for-program</th>
<th>Input objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Data</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$i$</td>
<td>$i+1$</td>
</tr>
<tr>
<td>$i+1$</td>
<td>$i+1$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Definition.** 0 is a type; $\sigma \rightarrow \tau$ is a type if $\sigma$ and $\tau$ are types; $\sigma \times \tau$ is a type if $\sigma$ and $\tau$ are types. We say a type $\tau$ has degree $n$ when $dg(\sigma) = n$ where $dg(0) = 0$; $dg(\sigma \rightarrow \tau) = \max(dg(\sigma) + 1, dg(\tau))$; and $dg(\sigma \times \tau) = \max(dg(\sigma), dg(\tau))$. We define the cardinality of type $\sigma$ at base $b$, in symbols $|\sigma|_b$, by recursion on the build-up of $\sigma$: $|0|_b = b$; $|\rho \rightarrow \tau|_b = |\rho|_b^b$; and $|\rho \times \tau|_b = |\rho|_b \times |\tau|_b$. □

Lemma 1 still holds for our new definitions. Besides, clause (i) of the lemma now also holds for the case $n = 0$.

**Definition.** We define the terms of the standard typed $\lambda$-calculus.

- (Variables.) We have an infinite supply of variables $x_0^\sigma, x_1^\sigma, x_2^\sigma, \ldots$ for each type $\sigma$. A variable of type $\sigma$ is a term of type $\sigma$.
- (\lambda-abstraction.) $\lambda x M$ is a term of type $\sigma \rightarrow \tau$ if $\sigma$ is a variable of type $\sigma$ and $M$ is a term of type $\tau$.
- (Application.) $(MN)$ is a term of type $\tau$ if $M$ is a term of type $\sigma \rightarrow \tau$ and $N$ is a term of type $\sigma$.
- (Product.) $(M, N)$ is a term of type $\sigma \times \tau$ if $M$ is a term of type $\sigma$ and $N$ is a term of type $\tau$.
- (Projections.) $\text{fst}M$ (\text{snd}M) is a term of type $\sigma (\tau)$ if $M$ is a term of type $\sigma \times \tau$.

The reduction rules of the standard typed $\lambda$-calculus are $(\lambda x M)N \vdash M[x := N]$ (\beta-conversion); $\text{fst}(M, N) \vdash M$; and $\text{snd}(M, N) \vdash N$. We will call all the three standard reduction rules for $\beta$- conversions. (This might depart from the standard terminology.) We will work in extensions of the standard typed $\lambda$-calculus, and $\vdash^\ast$ denotes the reflexive and transitive closure of the (extended) reducibility relation.
If $M \triangleright * N$ or $N \triangleright * M$, we will say that the two terms are equal and use the notation $M = N$. When it is not possible to use $\beta$-conversion on a term, we will say that the term is on $\beta$-normalform.

We will assume that the reader is familiar with the typed $\lambda$-calculus, and we will use the standard conventions from the literature. E.g. we omit parentheses and $MNPQ$ should be read $(((MN)P)Q)$; $F(X,Y)$ means $(FX)Y$; $\lambda xyz.M$ means $\lambda x(\lambda y(\lambda z.M))$. Occasionally, we will indicate the types of terms and variables by superscripts, e.g. $\lambda x^0 y^\tau.M^\tau$ indicates that the variables $x$ and $y$ have, respectively, type $0$ and $\tau$, and that the term $M$ has type $\tau$. For more on the $\lambda$-calculus see e.g. [7].

**Definition.** The calculus $T^-$ is the standard typed $\lambda$-calculus extended with the constant $1:0$, and for each type $\sigma$ the recursor $R_\sigma$ of type $\sigma,0 \rightarrow \sigma,0 \rightarrow \sigma$.

The calculus $T$ is the calculus $T^-$ extended with the constants $0:0$ (zero) and $s:0 \rightarrow 0$, the reduction rule $1 \triangleright s0$, and for each type $\sigma$, the reduction rules

$$R_\sigma(P,Q,0) \triangleright P \quad R_\sigma(P,Q,sN) \triangleright Q(N,R_\sigma(P,Q,N)).$$

We use $\pi$ to denote the numeral $s^00$ where $s^00 = 0$ and $s^{n+1}0 = (ss^n0)$.

Note that the calculus $T^-$ has no reduction rules in addition to those of the standard typed $\lambda$-calculus, and that e.g. the term $R_\sigma(M,N,1)$ is irreducible in the calculus $T^-$ if $M$ and $N$ are irreducible.

It is well known that any closed $T$-term of type $0$ normalizes to a unique numeral. Thus, a closed term $M$ of type $0 \rightarrow 0$ defines a function $f : \mathbb{N} \rightarrow \mathbb{N}$, and the value $f(n)$ can be computed by normalizing the term $M\pi n$. 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. This is not the case, and in [11] the authors show that a $0$-$1$ valued function is definable in $T^-$ if it is Kalmár elementary. In the following we will see that the $T^-$-calculus also induces a very interesting complexity-theoretic hierarchy.

**Definition.** The term $M:0$ defines the problem $A$ when $M\pi = \overline{0} \iff n \in A$. The rank $\text{rk}(M)$ of the $T$-term $M$ equals the least $n \in \mathbb{N}$ such that for any recursor $R_\sigma$ occurring in $M$ we have $\text{deg}(\sigma) \leq n$. A problem $A$ belongs to the class $\mathcal{F}^i$ if $A$ can be defined by a $T^-$-term of rank $\leq i$. It is trivial that $\mathcal{F}^i \subseteq \mathcal{F}^{i+1}$, and we have the functional hierarchy and $\bigcup_{i \in \mathbb{N}} \mathcal{F}^i$. In the remainder of this paper a functional is simply a closed $T^-$-term. (This is of course abuse of language.) If nothing else is said, the capital letters $F,G,H$ with or without decorations, denote closed $T$-terms. (Note that in our terminology, every functional is a closed $T$-term, but some closed $T$-terms will not be functionals.)

We have the following characterization of the alternating space-time hierarchy $\bigcup_{i \in \mathbb{N}} \text{STLIN}^i$. 

\[\]
Theorem 4 (Main Theorem). SPACE $2^i = \mathcal{F}^{2i}$ and TIME $2^{i+1} = \mathcal{F}^{2i+1}$.

The characterization invites to a further study of the hierarchy in a term rewriting framework. Such a study is undertaken in Bara [2].

The remainder of this paper is solely dedicated to the proof of The Main Theorem. In Subsection 5.2 we prove SPACE $2^i \subseteq \mathcal{F}^{2i}$ and TIME $2^{i+1} \subseteq \mathcal{F}^{2i+1}$; in Subsection 5.3 we prove $\mathcal{F}^{2i} \subseteq$ SPACE $2^i$ and $\mathcal{F}^{2i+1} \subseteq$ TIME $2^{i+1}$. It is possible to read Subsection 5.3 without reading 5.2.

5.2 We prove SPACE $2^i \subseteq \mathcal{F}^{2i}$ and TIME $2^{i+1} \subseteq \mathcal{F}^{2i+1}$

Lemma 12 (Basic functions). The following number-theoretic functions can be defined by $T^-$-terms of rank 0. (i) 0, 1 (constant functions); (ii) for each fixed $k \in \mathbb{N}$ the function $C_k(x)$ where $C_k(x) = k$ if $x \geq k$, and $C_k(x) = x$ otherwise (so $C_k(x)$ is the almost everywhere constant function yielding $k$ for all but finitely many values of $x$); (iii) $P(x)$ (predecessor) (iv); $x - y$ (modified subtraction); (v) $f(x, y, z)$ such that $f(x, y, z) = x \oplus y + 1$ when $x = y$; (vi) $c(x, y, z)$ where $c(x, y, z) = x$ if $z = 0$ and $c(x, y, z) = y$ if $z \neq 0$; (vii) $\max(x, y)$.

Proof. The constant function 1 is defined by the initial $T^-$-term 1. The projection function $u^i(x_1, \ldots, x_n) = x_i$ 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 and 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 (iii) holds. (iv) holds since we have $x - 0 = x$ and $x - (y + 1) = P(x - y)$. (v) holds since $c(x, y, 0) = x$ and $c(x, y, z + 1) = y$. (vi) holds since $\max(x, y) = c(x, y, 1 - (x - y))$. (vii) holds since $x \oplus y + 1 = c(0, m - (m - z) - 1, m - z)$. It remains to prove that (ii) holds. Let $M^0(z) = 0$ and $M^{n+1}(z) = M^n(z) \oplus z + 1$. We can define $M^n$ for any fixed $n \in \mathbb{N}$. Further, $M^n(z) = n \mod z + 1$. Thus, we have $C_k(x) = c(x, M^k(x), P^k(x))$ where $P^k$ is the predecessor function repeated $k$ times. Hence, (ii) holds.

Lemma 13 (Conditional functionals). For any type $\sigma$ there exists a functional $\text{Cond}_\sigma : 0, \sigma, \sigma \to \sigma$ such that

$$\text{Cond}_\sigma(\overline{n + 1}, F, G) = \begin{cases} F & \text{if } n = 0 \\ G & \text{otherwise.} \end{cases}$$

Moreover, $\text{rk}(\text{Cond}_\sigma) = \text{dg}(\sigma)$.

Proof. Let $\text{Cond}_\sigma \equiv \lambda x^0 y^\sigma z^\sigma. R_\sigma(y^\sigma, \lambda u^0 v^\sigma.z^\sigma, x^0)$. Then $\text{Cond}_\sigma(\overline{n + 1}, F, G) = R_\sigma(F, \lambda u^0 v^\sigma.G, \overline{n + 1}) = F$ and

$$\text{Cond}_\sigma(\overline{n + 1}, F, G) = R_\sigma(F, \lambda u^0 v^\sigma.G, \overline{n + 1}) = \lambda u^0 v^\sigma.G(\overline{n}, R_\sigma(F, \lambda u^0 v^\sigma.G, \overline{n})) = G.$$
Furthermore, $R_\sigma$ is the only recursor in $\text{Cond}_\sigma$, and thus we have $\text{rk}(\text{Cond}_\sigma) = \text{dg}(\sigma)$. □

**Lemma 14 (Iteration functionals).** For all types $\sigma$ and $\tau$ there exists a functional $\text{It}^\tau_\sigma : 0, \tau \rightarrow \tau, \tau \rightarrow \tau$ such that $\text{It}^\tau_\sigma(\vec{F}, G) = F^{[\sigma][T+1]}(G)$. Moreover, $\text{rk}(\text{It}^\tau_\sigma) = \text{dg}(\sigma) + \text{dg}(\tau)$.

**Proof.** We prove the lemma by induction on the build-up of $\sigma$. Assume $\sigma = 0$. Let

$$\text{It}^\tau_0 \equiv \lambda n^0 Y^{\tau \rightarrow \tau} X^n. R_\tau(Y(X), \lambda x^0.Y, n) .$$

Obviously, we have $\text{rk}(\text{It}^\tau_0) = \text{dg}(0) + \text{dg}(\tau)$. We prove by induction on $\ell$ that $\text{It}^\tau_{\ell}(\vec{F}, G) = F^{\ell+1}(G)$. We have $\text{It}^\tau_{\ell}(0, F, G) = R_\tau(F(G), \lambda x.F, 0) = F(G)$. By the induction hypothesis we have

$$\text{It}^\tau_{\ell}(\vec{F} + 1, F, G) = R_\tau(F(G), \lambda x.F, \vec{F} + 1) = ((\lambda x.F)\vec{F})R_\tau(F(G), \lambda x.F, \vec{F}) =
((\lambda x.F)\vec{F})\text{It}^\tau_{\ell}(\vec{F}, F, G) = FF^{\ell+1}(G) = F^{\ell+2}(G) .$$

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

Assume $\sigma = \sigma_1 \times \sigma_2$. By the induction hypothesis we have functionals $\text{It}^\tau_{\sigma_1}$ and $\text{It}^\tau_{\sigma_2}$ satisfying the lemma. Define $\text{It}^\tau_{\sigma}$ such that

$$\text{It}^\tau_{\sigma}(\vec{F}, G) = \text{It}^\tau_{\sigma_1}(\vec{F}, \text{It}^\tau_{\sigma_2}(\vec{F}, G), G) .$$

The rank of $\text{It}^\tau_{\sigma}$ will equal the maximum of the rank of $\text{It}^\tau_{\sigma_1}$ and the rank of $\text{It}^\tau_{\sigma_2}$. Thus,

$$\text{rk}(\text{It}^\tau_{\sigma}) = \max(\text{rk}(\text{It}^\tau_{\sigma_1}), \text{rk}(\text{It}^\tau_{\sigma_2})) \quad \text{def. of rk}$$

$$= \max(\text{dg}(\sigma_1) + \text{dg}(\tau), \text{dg}(\sigma_2) + \text{dg}(\tau)) \quad \text{ind. hyp.}$$

$$= \max(\text{dg}(\sigma_1), \text{dg}(\sigma_2)) + \text{dg}(\tau)$$

$$= \text{dg}(\sigma_1 \times \sigma_2) + \text{dg}(\tau) \quad \text{def. of dg}$$

$$= \text{dg}(\sigma) + \text{dg}(\tau) . \quad \sigma = \sigma_1 \times \sigma_2$$

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

$$\text{It}^\tau_{\sigma}(\vec{F}, F, G) = \text{It}^\tau_{\sigma_1}(\vec{F}, \text{It}^\tau_{\sigma_2}(\vec{F}, F), G) \quad \text{def. of } \text{It}^\tau_{\sigma}$$

$$= \text{It}^\tau_{\sigma_1}(\vec{F}, F)^{[\sigma][T+1]}(G) \quad \text{ind. hyp. on } \sigma_1$$

$$= F^{[\sigma][T+1]}(G) \quad \text{ind. hyp. on } \sigma_2$$

$$= F^{[\sigma][T+1]}(G) . \quad \text{def. of } |\sigma|_{\ell+1}$$

Assume $\sigma = \sigma_1 \rightarrow \sigma_2$. By the induction hypothesis we have functionals $\text{It}^\tau_{\sigma_2}$ and $\text{It}^\tau_{\sigma_1 \rightarrow \tau}$ satisfying the lemma. Define $\text{It}^\tau_{\sigma}$ from $\text{It}^\tau_{\sigma_2}$ and $\text{It}^\tau_{\sigma_1 \rightarrow \tau}$ such that

$$\text{It}^\tau_{\sigma}(\vec{F}, F, X) = \text{It}^\tau_{\sigma_1}(\vec{F}, \text{It}^\tau_{\sigma_2}(\vec{F}, F), X) .$$
First we prove that the rank of \( \text{rk} (\text{It}_\tau^\sigma) = \text{dg}(\sigma) + \text{dg}(\tau) \). Obviously, the rank of \( \text{It}_\sigma^\tau \) will be the maximum of the rank of \( \text{It}_{\sigma_1}^{\tau_1} \) and the rank of \( \text{It}_{\sigma_2}^{\tau_2} \). Hence,

\[
\text{rk}(\text{It}_\tau^\sigma) = \max(\text{rk}(\text{It}^{\tau_1 \rightarrow \tau_{\sigma_1}}), \text{rk}(\text{It}^{\tau_2 \rightarrow \tau_{\sigma_2}})) \\
= \max(\max(\text{dg}(\sigma_1) + \text{dg}(\tau \rightarrow \tau), \text{dg}(\sigma_2) + \text{dg}(\tau)), \text{ind. hyp.}) \\
= \max(\max(\text{dg}(\sigma_1) + \text{dg}(\tau) + 1, \text{dg}(\sigma_2) + \text{dg}(\tau)), \text{def. of dg}) \\
= \max(\max(\text{dg}(\sigma_1) + 1, \text{dg}(\sigma_2)) + \text{dg}(\tau)) \\
= \text{dg}(\sigma) + \text{dg}(\tau) .
\]

We will now prove that we indeed have

\[
\text{It}_\tau^\sigma(F, F, X) = F^{\text{dg}(\tau + 1)}(X) \quad \text{(Goal)}
\]

Let \( A \) abbreviate \( \text{It}_\tau^\sigma(F) \). Hence, we have

\[
A(F, X) = F^{\text{dg}(\tau + 1)}(X) \quad \text{(\dag)}
\]

by the induction hypothesis. We need

\[
A^k(F, Y) = F^{\text{dg}(\tau + 1)}(Y) \quad \text{(Claim)}
\]

for any \( Y \) of type \( \tau \). (Goal) follows from (Claim) since

\[
\text{It}_\tau^\sigma(F, F, X) = \text{It}^{\tau_1 \rightarrow \tau_2}_\sigma(F, X) \quad \text{def. of It}_\tau^\sigma \\
= A^{\text{dg}(\tau_1 + 1)}(F)(X) \quad \text{ind. hyp. on } \sigma_1 \\
= F^{\text{dg}(\tau + 1)}(X) \quad \text{(Claim)} \\
= F^{\text{dg}(\tau + 1)}(X) . \quad \text{def. of } \text{dg}(\tau + 1)
\]

(Claim) is proved by induction on \( k \). We skip the details.

\( \square \)

**Definition.** Let \( \mathcal{V} \) be a valuation, that is a set of pairs \( x/v \) where \( x/v \) is interpreted as the variable \( x : \sigma \) is assigned the numerical value \( v < |\sigma|_b \) (the base \( b \) will be understood from the context). For any \( T \)-term \( M \) we define the value of \( M \) at the base \( b \) under valuation \( \mathcal{V} \), in symbols \( \text{val}_b^\mathcal{V}(M) \). (Note that we have \( \text{val}_b^\mathcal{V}(M) < |\sigma|_b \) for any closed term \( M : \sigma \)). Recall that for numbers \( a : (\rho \rightarrow \tau)_b \) and \( i : \rho_b \) we denote the \( i \)’th digit in \( a \) by \( a[i]_b \) (see the definition in Section 2).

- Let \( \text{val}_b^\mathcal{V}(x) = v \) if \( x \) is a variable and \( x/v \in \mathcal{V} \).
- Let \( \text{val}_b^\mathcal{V}(1) = 1 \).
- Let \( \text{val}_b^\mathcal{V}(0) = 0 \).
- Let \( \text{val}_b^\mathcal{V}(s M) = \text{val}_b^\mathcal{V}(M) + 1 \) (mod \( b + 1 \)).
- Let \( \text{val}_b^\mathcal{V}((M N)) = \text{val}_b^\mathcal{V}(M)[\text{val}_b^\mathcal{V}(N)]_b \).
- Let

\[
\text{val}_b^\mathcal{V}(\lambda x^\sigma M\tau) = \sum_{i < |\tau|_b} \text{val}_b^\mathcal{V}(M) \times |\tau|_b^i
\]

where \( \mathcal{V}' = \mathcal{V} \cup \{x/i\} \).
− Let $$\text{val}^\Gamma_b((M^\sigma, N^\tau)) = \text{val}^\Gamma_b(M) \times |\tau|_b + \text{val}^\Gamma_b(N)
\text{.}
$$
− Recall that $$R^\sigma$$ has type $$\sigma, 0 \rightarrow \sigma \rightarrow \sigma, 0 \rightarrow \sigma$$, let $$\rho = 0 \rightarrow \sigma \rightarrow \sigma$$, and let

$$\text{val}^\Gamma_b(R^\sigma) = \sum_{w < |\sigma|_b} |\sigma|_b^w \times (\sum_{w < |\rho|_b} |0 \rightarrow \sigma|_b^w \times (\sum_{n < |0|_b} |\sigma|_b^n \times \gamma^n))$$

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

We will occasionally write $$\text{val}_b$$ in place of $$\text{val}^\Gamma_b$$.

Note that when $$F$$ is a closed $$T$$-term, the value $$\text{val}_b(F)$$ is defined. Further, recall that when nothing else is said, the symbols $$F, G, H$$, possibly decorated, denote close $$T$$-terms.

**Lemma 15 (Equality functionals).** For any type $$\sigma$$ there exists an equality functional $$\text{Eq}_\sigma : 0, \sigma, \sigma \rightarrow 0$$ such that

$$\text{Eq}_\sigma(\tau, F, G) = \overline{0} \text{ if and only if } \text{val}_{t+1}(F) = \text{val}_{t+1}(G) \text{.}$$

Moreover, we have $$\text{rk}(\text{Eq}_\sigma) \leq 2\text{dg}(\sigma)^{-2}$$.

**Proof.** For any type $$\sigma$$ we will define functionals $$0_\sigma : 0$$ and $$\text{Suc}_\sigma : 0, \sigma \rightarrow \sigma$$ such that $$\text{val}_{t+1}(0_\sigma) = 0$$ and

$$\text{val}_{t+1}(F) + 1 = \text{val}_{t+1}(\text{Suc}_\sigma(\tau, F)) \text{ (mod } |\sigma|_{t+1}) \text{.}$$

Further, for any type $$\sigma$$ we will define functionals $$\text{Le}_\sigma : 0, \sigma, \sigma \rightarrow 0$$ and $$\text{Eq}_\sigma : 0, \sigma, \sigma \rightarrow 0$$ such that

$$\text{Le}_\sigma(\tau, F, G) = 0 \text{ if and only if } \text{val}_{t+1}(F) \leq \text{val}_{t+1}(G)$$

$$\text{Eq}_\sigma(\tau, F, G) = 0 \text{ if and only if } \text{val}_{t+1}(F) = \text{val}_{t+1}(G) \text{.}$$

We will see that $$\text{rk}(0_\sigma) = 0$$, $$\text{rk}(\text{Suc}_\sigma) \leq 2\text{dg}(\sigma)^{-2}$$, $$\text{rk}(\text{Le}_\sigma) \leq 2\text{dg}(\sigma)^{-2}$$ and $$\text{rk}(\text{Eq}_\sigma) \leq 2\text{dg}(\sigma)^{-2}$$.

We define the functionals inductively over the build-up of $$\sigma$$. It follows from Lemma 12 that the functionals can be defined when $$\sigma = 0$$.

Assume that $$\sigma = \pi \rightarrow \tau$$ by the induction hypothesis, all the functionals are defined for $$\pi$$ and $$\tau$$. First we define $$0_\sigma \equiv \lambda x^\pi . 0_\tau$$. Obviously we have $$\text{rk}(0_\sigma) = 0$$.

Next we define $$F : 0, \sigma, \sigma, (0, \pi) \rightarrow (0, \pi)$$ such that

$$F(\ell, X, Y, \langle i, j \rangle) = \begin{cases} \langle i, \text{Suc}_\sigma(j) \rangle & \text{if } \text{Eq}_\sigma(\ell, X(j), Y(j)) = 0 \\ \langle 0, \text{Suc}_\sigma(j) \rangle & \text{if } \text{Eq}_\sigma(\ell, X(j), Y(j)) > 0 \text{ and } \text{Le}_\sigma(\ell, X(j), Y(j)) = 0 \\ \langle 1, \text{Suc}_\sigma(j) \rangle & \text{otherwise.} \end{cases}$$
Hence, let $F \equiv$
\[
\lambda \ell \lambda X : \pi \to Y : \pi \to z : 0 \times \pi .
\]
\[
\text{Cond}_{0 \times \pi}(\text{Eq}_\tau(\ell, X(snd z), Y(snd z)), \langle \text{fst}_z, \text{Succ}_\pi(snd z) \rangle),
\]
\[
\text{Cond}_{0 \times \pi}(\text{Le}_\tau(\ell, X(snd z), Y(snd z)), \langle 0, \text{Succ}_\pi(snd z) \rangle),
\]
\[
\langle 1, \text{Succ}_\pi(snd z) \rangle .
\]

Then, let $\text{Le}_\sigma \equiv \lambda \ell \lambda X. \lambda Y. \text{fst}_1 \lambda \ell \lambda x, (\ell, F(\ell, X, Y), \langle 0, 0, \sigma \rangle)$ and

$\text{Eq}_\sigma \equiv \lambda \ell \lambda X. \text{Cond}_{0}(\text{Le}_\sigma(\ell, X, Y), \text{Cond}_{0}(\text{Le}_\sigma(\ell, Y, X), 0, 1, 1) .
$

We have

\[
\text{rk}(F) = \max(\text{rk}(\text{Cond}_{0 \times \pi}), \text{rk}(\text{Eq}_\tau), \text{rk}(\text{Succ}_\pi), \text{rk}(\text{Le}_\tau)) \quad \text{def. of rk, def. of F}
\]

\[
= \max(\text{rk}(\text{dg}(0 \times \pi), \text{rk}(\text{Eq}_\tau), \text{rk}(\text{Succ}_\pi), \text{rk}(\text{Le}_\tau)) \quad \text{Lemma 13}
\]

\[
\leq \max(\text{dg}(\pi), 2 \text{dg}(\pi) - 2, 2 \text{dg}(\tau) - 2)
\]

\[
\text{ind. hyp. def. of dg}
\]

Further, we have

\[
\text{rk}(\text{Eq}_\sigma) = \text{rk}(\text{Le}_\sigma)
\]

\[
= \max(\text{rk}(F), \text{rk}(\text{Le}_\sigma)) \quad \text{def. of rk, def. of Eq}_\sigma
\]

\[
= \max(\text{rk}(F), \text{dg}(\pi) + \text{dg}(0 \times \pi)) \quad \text{Lemma 14}
\]

\[
\leq \max(\text{dg}(\pi), 2 \text{dg}(\pi) - 2, 2 \text{dg}(\tau) - 2, 2 \text{dg}(\pi))
\]

\[
\text{the bound on } \text{rk}(F)
\]

\[
= \max(2 \text{dg}(\tau) - 2, 2 \text{dg}(\pi))
\]

\[
= \max(2 \text{dg}(\tau) - 1, 2 \text{dg}(\pi))
\]

\[
= 2 \max(\text{dg}(\pi), \text{dg}(\tau) - 1)
\]

\[
= 2 \max(\text{dg}(\pi) + 1, \text{dg}(\tau)) - 1
\]

\[
= 2(\text{dg}(\pi) - 1)
\]

\[
= 2 \text{dg}(\sigma) - 2 .
\]

Thus, $\text{Eq}_\sigma$ and $\text{Le}_\sigma$ have the required rank. In order to define $\text{Succ}_\sigma$, we need to define the “carry” functional $C : 0, (\sigma \to \tau) \to \tau$. First we define the functional $G : 0, \sigma, (0, \pi) \to (0, \pi)$ such that

\[
G(\ell, X, \langle i, j \rangle) = \begin{cases} 
\langle 0, \text{Succ}_\pi(j) \rangle & \text{if } i = 0 \text{ and } \text{Succ}_\tau(\ell, X(j)) = 0_

\langle 1, j \rangle & \text{otherwise.}
\end{cases}
\]

Hence, let $G \equiv$

\[
\lambda \ell \lambda X : \pi \to z : 0 \times \pi . \text{Cond}_{0}(\text{fst}_z, \text{Cond}_{0 \times \pi}(\text{Eq}_\tau(\ell, \text{Succ}_\pi(X(snd z)), 0_\tau),
\]

\[
\langle 0, \text{Succ}_\pi(snd z) \rangle, \langle 1, \text{Succ}_\pi(snd z) \rangle, \langle 1, \text{Succ}_\pi(snd z) \rangle .
\]

Further, let $C_\sigma \equiv \lambda \ell \lambda X. \lambda \text{snd}_1 \lambda \ell \lambda x, (\ell, G(\ell, X), \langle 0, 0, \sigma \rangle).$ The functional $C_\sigma$ yields the “carry borderline” for the successor $\text{Succ}_\sigma$. Thus, let

\[
\text{Succ}_\sigma \equiv \lambda \ell \lambda x. \lambda \text{snd}_i \lambda \text{Le}_\sigma(\ell, i, C_\sigma(\ell, X)), \text{Succ}_\tau(X(i)), X(i) .
\]
By an argument similar to the one showing that the ranks of $E_{\sigma}$ and $L_{\sigma}$ are bounded by $2d(\sigma)^{-2}$, we can show that the rank of $C_{\sigma}$ also is bounded by $2d(\sigma)^{-2}$. It is easy to see that the rank of $\text{Suc}_{\sigma}$ equals the rank of $C_{\sigma}$. Thus, $\text{Suc}_{\sigma}$ has the required rank.

Assume that $\sigma = \pi \times \tau$ the functionals are defined for $\pi$ and $\tau$. Let $0_{\sigma} \equiv (0_{\pi}, 0_{\tau})$. Define $\text{Suc}_{\sigma}$ such that

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

Define $L_{\sigma}$ such that $L_{\sigma}(\ell, \langle F, G \rangle, \langle F', G' \rangle) = 0$ iff

$$(L_{\pi}(\ell, F, F') = 0 \land \text{Eq}_{\pi}(\ell, F, F') > 0) \lor
(L_{\pi}(\ell, G, G') = 0 \land \text{Eq}_{\pi}(\ell, F, F') = 0).$$

Define $\text{Eq}_{\sigma}$ such that $\text{Eq}_{\sigma}(\ell, F, G) = 0$ iff

$$L_{\pi}(\ell, F, G) = 0 \land L_{\pi}(\ell, G, F) = 0.$$

It is easy to construct functionals $\text{Suc}_{\sigma}$, $L_{\sigma}$ and $\text{Eq}_{\sigma}$ with the required properties

***

**Lemma 16 (Modification functionals).** For any types $\bar{\sigma} = \sigma_1, \ldots, \sigma_k$ and $\tau$ there exists a modification functional $\text{Md}_{\bar{\sigma} \rightarrow \tau} : 0, \bar{\sigma} \rightarrow \tau, \bar{\sigma} \rightarrow \bar{\sigma} \rightarrow \tau$ such that

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

Moreover, we have $\text{rk}(\text{Md}_{\bar{\sigma} \rightarrow \tau}) \leq \max(2 \max(\text{dg}(\sigma_1), \ldots, \text{dg}(\sigma_k))^{-2}, \text{dg}(\tau))$ (**).

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

$$\text{Md}_{\rho \rightarrow \tau} \equiv \lambda F^0 F^\rho \rightarrow \tau X^0 V^\tau Y^\rho . \text{Cond}_{\tau}(\text{Eq}_{\rho}(\ell, X, Y), V, F(Y)).$$

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

$$\text{Md}_{\rho, \bar{\pi} \rightarrow \tau} \equiv \lambda \ell F^0 F^\rho \rightarrow \tau X^0 Y^{\ell+1} \ldots X^k_{\bar{\pi}} Y^\tau .
\text{Md}_{\rho \rightarrow \tau}(\ell, F, X_0, \text{Md}_{\bar{\pi} \rightarrow \tau}(\ell, F(X_0), X_1, \ldots, X_k, V)).$$

It is easy to see that $\text{rk}(\text{Md}_{\bar{\sigma} \rightarrow \tau}) = \max(\text{rk}(\text{Eq}_{\sigma_1}), \ldots, \text{rk}(\text{Eq}_{\sigma_k}), \text{rk}(\text{Cond}_{\tau}))$. Thus, (***) holds by Lemma 13 and Lemma 15.

**Theorem 5.** We have $L^{i,j}_{\text{vir}} \subseteq F^{i+j}$ whenever $i \leq j \leq i + 1$. 

\text{\hfill \Box}
Proof. Let \( A \in \mathcal{L}_{x_i \in \sigma_i} \). Thus, there exists a program \( P \) of loop degree \( i \) and data degree \( j \) such that
\[
\forall x_2, \ldots, x_k \exists y_1, \ldots, y_{k-1} [ x_1, \ldots, x_k \{ P \} b y_1, \ldots, y_{k-1}, 0 ] \text{ iff } x_1 \in A
\]
where \( b = \max(x_1, 1) + 1 \). Assume that \( P \) is a program in the variables \( x_1, \ldots, x_k \) of type \( \sigma = \sigma_1, \ldots, \sigma_k \) respectively. Further, assume that the loops in \( P \) have the types \( \pi_1, \ldots, \pi_r \). Note that we have
\[
2 \max(\text{deg}(\sigma_1), \ldots, \text{deg}(\sigma_k)) - 2 \leq \max(\text{deg}(\pi_1), \ldots, \text{deg}(\pi_r)) + \max(\text{deg}(\sigma_1), \ldots, \text{deg}(\sigma_k))
\]
since \( i \leq j \leq i + 1 \).

We will prove that there exists a functional \( f : 0, \sigma \rightarrow \sigma_1 \) to the program variable \( x_i \). The required functional \( f \) is given by
\[
f(x) = F^p_k(\max(x_1, 1), x_1, G_2, \ldots, G_k)
\]
where \( G_2, \ldots, G_k \) are arbitrary constants of appropriate types. By Lemma 12 (viii), \( f \) has the same rank as \( F^p_k \).

We define \( F^p_1, \ldots, F^p_k \) recursively over the build-up of \( P \). For each case in the definition we will assure that
\[
\text{rk}(F^p_m) \leq \max(\text{deg}(\pi_1), \ldots, \text{deg}(\pi_r)) + \max(\text{deg}(\sigma_1), \ldots, \text{deg}(\sigma_k))
\]
holds for \( m = 1, \ldots, k \).

Assume \( P \equiv X_m \). Then \( X_m \) has type \( 0 \). By Lemma 12 we have a functional \( F^p_m \) of rank 0 such that \( F^p_m(\ell, X_m) = X_m \oplus_{\ell+1} 1 \). Let \( F^p_j \equiv \lambda \ell X_1 \ldots X_k X_j \) for \( j \neq m \). The functionals \( F^p_1, \ldots, F^p_k \) have rank 0 and satisfy (†) trivially.

Assume \( P \equiv X_m [X_{i_1}, \ldots, X_{i_r}] \) where \( X_m \) has type \( \beta \rightarrow 0 \). Let
\[
F^p_m = \lambda \ell \text{Md}_{\beta \rightarrow 0}(\ell, X_m, X_{i_1}, \ldots, X_{i_r}, X_m(X_{i_1}, \ldots, X_{i_r}) \oplus_{\ell+1} 1)
\]
where \( \text{Md}_{\sigma_1, \ldots, \sigma_r \rightarrow 0} \) is the modification functional given in Lemma 16. Let \( F^p_j \equiv \lambda \ell X_1 \ldots X_k X_j \) for \( j \neq m \). By Lemma 16 we have
\[
\text{rk}(\text{Md}_{\beta \rightarrow 0}) \leq 2 \max(\text{deg}(\sigma_1), \ldots, \text{deg}(\sigma_k)) - 2.
\]
Hence, (†) holds by (*).

Assume \( P \equiv Q \). Let
\[
F^p_m \equiv \lambda \ell \text{Md}_{\beta \rightarrow 0}(\ell, F^p_m(\ell), \ldots, F^p_m(\ell))
\]

for $m = 1, \ldots , k$. Assume that ($\dagger$) holds for $F_1^0, \ldots , F_k^0$ and $F_m^0$. Then ($\dagger$) trivially also holds for $F_m^p$.

Assume $P \equiv \text{fin } t \{Q\}$ where the test $t$ has the form $t \equiv x_i [X_1, \ldots, X_t]$. Let

$$F_m^p \equiv \lambda \ell X. \text{Cond}_{\sigma_m}(X_i(X_1, \ldots, X_t), X_m, F_m^0(\ell, X))$$

for $m = 1, \ldots, k$ where Cond$_{\sigma_m}$ is the conditional functional given in Lemma 13. We have $\text{rk(Cond}_{\sigma_m}) = \text{dg(} \sigma_m \text{)}$, and thus ($\dagger$) holds by the induction hypothesis.

Assume $P \equiv \text{for}_x \{Q\}$. Let $\text{pr}_x(X_1, \ldots, X_k) = X_i$. Obviously the functional $\text{pr}_x$ can be defined with rank 0. Let

$$F^q \equiv \lambda \ell X^{\sigma_1 \times \cdots \times \sigma_k} \langle F^0(\ell, \text{pr}_1 X, \ldots, \text{pr}_k X), \ldots, F^0(\ell, \text{pr}_1 X, \ldots, \text{pr}_k X) \rangle .$$

Further, let

$$F' \equiv \lambda \ell X^{\sigma_1 \times \cdots \times \sigma_k} \text{It}_{\xi}^{\tau} (\ell, F^0(\ell), X)$$

where It$_{\xi}^{\tau}$ is the iteration functional given in Lemma 14. Finally let

$$F_m^p \equiv \lambda \ell X. \text{pr}_m(F'(\ell, (X_1, \ldots, X_k)))$$

for $m = 1, \ldots, k$. By Lemma 14 we have $\text{rk(F')} = \text{dg(}\xi\text{)} + \text{dg(}\sigma_1 \times \cdots \times \sigma_k\text{)}. The rank of $F_m^p$ equals the rank of $F'$, and thus, ($\dagger$) holds. This completes the proof of the theorem. □

**Corollary 1.** We have space $2_i^{=\omega} \subseteq \mathcal{F}^{2i}$ and time $2_{i+1}^{=\omega} \subseteq \mathcal{F}^{2i+1}$ for any $i \in \mathbb{N}$.

**Proof.** This follows from Theorem 5 and Theorem 3. □

**5.3 We prove $\mathcal{F}^{2i} \subseteq \text{space } 2_i^{=\omega}$ and $\mathcal{F}^{2i+1} \subseteq \text{time } 2_{i+1}^{=\omega}$**

**Definition.** The $\lambda[\cdot\cdot\cdot]$-calculus is the standard typed $\lambda$-calculus extended with the constants $0 : 0$ (zero), $s : 0 \rightarrow 0$ (successor) and for each type $\sigma$ the **bracket** $[\cdot\cdot\cdot]_\sigma$ of type $0, 0, \sigma \rightarrow \sigma$. (We use mixfix notation for legibility’s sake: $[M^0, N^0, P^0, Q^0]_\sigma$ is a term of type $\sigma$.) In addition to the standard reduction rules for the typed $\lambda$-calculus, the $\lambda[\cdot\cdot\cdot]$-calculus embodies the reduction rules $[0, N, P, Q] \triangleright P$; $[sM, 0, P, Q] \triangleright Q$; $[sM, sN, P, Q] \triangleright [M, N, P, Q]$; and

$$[M, N, P^{\sigma \rightarrow \tau}, Q^{\rho \rightarrow \tau}]_{\rho \rightarrow \tau} R^\sigma \triangleright [M, N, PR, QR]_{\tau} .$$

The **level** lev($M$) of the $\lambda[\cdot\cdot\cdot]$-term $M$ equals the least $n \in \mathbb{N}$ such that for any subterm $N : \sigma$ of $M$ we have $\text{dg(}\sigma\text{)} \leq n$.

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 $M$. □

The bracket permits if-then-else constructions. The term $[\pi, \pi, P, Q]$ reduces to $P$ if $m \leq n$; otherwise the term reduces to $Q$. 

---

Lars Kristiansen and Paul J. Voda
Definition. Let \( \mathcal{V} \) be a valuation, that is a set of pairs \( x/v \) where \( x/v \) is interpreted as the variable \( x : \sigma \) is assigned the numerical value \( v < |\sigma|_b \) (the base \( b \) will be understood from the context). For any \( \lambda \beta \)-term \( M \) we define the **value of \( M \) at the base \( b \) under valuation \( \mathcal{V} \)**, in symbols \( \text{val}^\mathcal{V}_b(M) \). (Note that we have \( \text{val}^\mathcal{V}_b(M) < |\sigma|_b \) for any closed term \( M : \sigma \).) Recall that for numbers \( a : (\rho \rightarrow \tau)_b \) and \( i : \rho_b \) we denote the \( i \)’th digit in \( a \) by \( a[i]_b \) (see the definition in Section 2).

- Let \( \text{val}^\mathcal{V}_b(x) = v \) if \( x \) is a variable and \( x/v \in \mathcal{V} \).
- Let \( \text{val}^\mathcal{V}_b(0) = 0 \).
- Let \( \text{val}^\mathcal{V}_b(sM) = \text{val}^\mathcal{V}_b(M) + 1 \) (mod \( b + 1 \)).
- Let \( \text{val}^\mathcal{V}_b((MN)) = \text{val}^\mathcal{V}_b(M) \text{val}^\mathcal{V}_b(N)_b \).
- Let

\[
\text{val}^\mathcal{V}_b(\lambda x^\sigma M^\tau) = \sum_{i < |\tau|_b} \text{val}^\mathcal{V}_b(M) \times |\tau|_b
\]

where \( \mathcal{V}' = \mathcal{V} \cup \{x/i\} \).
- Let \( \text{val}^\mathcal{V}_b((M^\sigma, N^\tau)) = \text{val}^\mathcal{V}_b(M) \times |\tau|_b + \text{val}^\mathcal{V}_b(N) \).
- Let

\[
\text{val}^\mathcal{V}_b([M, N, P, Q]) = \begin{cases} 
\text{val}^\mathcal{V}_b(P) & \text{if } \text{val}^\mathcal{V}_b(M) \leq \text{val}^\mathcal{V}_b(N) \\
\text{val}^\mathcal{V}_b(Q) & \text{otherwise.}
\end{cases}
\]

We will write \( \text{val}_b \) in place of \( \text{val}^\mathcal{V}_b \).

Lemma 17. Let \( M : 0 \) be a closed \( \lambda \beta \)-term of level \( k + 2 \).

- (i) There exists constant \( c \in \mathbb{N} \) such that the value \( \text{val}_{n+1}(M) \) can be computed in space \( \#M (\log_2 \#M)(2^{c|n|}) \).
- (ii) There exist a fixed polynomial \( p \) and a constant \( c \in \mathbb{N} \) such that the value \( \text{val}_{n+1}(M) \) can be computed in time \( p(\#M)2^{c|n|} \).

Proof. In this proof we will call a redex \((M^\sigma N)\) maximal if \( \text{deg}(\sigma) = k + 2 \). A maximal redex will be on one of the two forms

(1) \( (\lambda x M N) \)  
(2) \( ([M, N, P, Q] R) \)

where the types of \( \lambda x M \) and \([M, N, P, Q] \) have degree \( k + 2 \). In this proof we also need the notion of a semi-reduction. In a semi-reduction we do not replace a variable by a term as we e.g. do in an ordinary \( \beta \)-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 \( M^x_p \) denote the term we get when each occurrence the variable \( x \) in \( M \) is replaced by the pointer \( p_i \). (We use \( p_0, p_1, p_2, \ldots \) to denote pointers. The Turing machine constructed below will use binary numbers to represent the pointers.)

We will say that a string on the form

\[
C\{(\lambda x PQ)]/p_i : M_1/p_2 : M_2/ \ldots /p_t : M_t
\]


where $C\{(\lambda x P Q)\}$ is a $\lambda[$]-term possibly containing pointers, $(\lambda x P Q)$ is a redex in $C\{(\lambda x P Q)\}$, and $M_1, \ldots, M_t, P, Q$ are $\lambda[$]-terms possibly containing pointers, semi-reduces to the string

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

Likewise we say that a string on the form

$$C\{[M, N, P, Q]_R\}/p_1 : M_1/p_2 : M_2/\ldots/p_\ell : M_\ell$$

semi-reduces to the string

$$C\{[M, N, P_{\ell+1}, Q_{\ell+1}]\}/p_1 : M_1/p_2 : M_2/\ldots/p_\ell : M_\ell/p_{\ell+1} : R.$$ 

Let $M:0$ be a closed $\lambda[$]-term of level $k+2$. We will now construct a Turing machine computing $\text{val}_{n+1}(M)$. Let $w_0$ be the string of symbols given by $M$. 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 $(M N)$ such that there are no maximal redices inside $N$. Thereafter it semi-reduces $w_1$ to $w_2$ following the same procedure, then $w_2$ to $w_3$, an so on. Sooner or later, let us say after 17 steps, the process will terminate since there will be no maximal redices left. The string $w_{17}$ has the form $P/p_1 : M_1/p_2 : M_2/\ldots/p_{17} : M_{17}$ and represents a $\lambda[$]-term of level $k+1$. (Note that no semi-reductions will take place inside the terms $M_1, \ldots, M_{17}$ because these terms do not contain maximal redices.) 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.

Let $Q$ be the term of level $k+1$ represented by $w_{17}$. We have $\text{val}_{n+1}(Q) = \text{val}_{n+1}(M)$. The Turing machine will now compute $\text{val}_{n+1}(Q)$ using registers. A register is nothing but a marked area on one of the tapes dedicated to store a natural number. The function $\text{val}_{n+1}$ is defined inductively over the build-up of $\lambda[$]-terms. The Turing machine will compute the value $\text{val}_{n+1}(S)$ 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 $\text{val}_{n+1}(S)$. E.g. to compute the value $\text{val}_{n+1}(\lambda x S R)$, the Turing machine can first compute $a = \text{val}_{n+1}(\lambda x S)$, store $a$ in a register, thereafter compute $b = \text{val}_{n+1}(R)$, store $b$ in a register, and finally compute the value $\text{val}_{n+1}(\lambda x S R)$ by computing the number $\lambda[b]_{n+1}$. The Turing machine will only depart from this natural recursive procedure when it encounters subtrees of level $k+1$. There will be some such subterms since $Q$ has level $k+1$. For the sake of the argument assume that such a subterm has form $\lambda x R$ where $\text{lev}(R) = k$. In such a case the Turing machine will compute the value $\text{val}_{n+1}(\lambda x R S)$ by first computing $a = \text{val}_{n+1}(S)$ and then compute the value $\text{val}_{n+1}(R)$ where $\mathcal{V} = \mathcal{V} \cup \{x/a\}$. By following this pattern the Turing machine avoids using registers for storing numbers of type $\sigma$ where $\text{dg}(\sigma) = k+1$.

We will now argue that the Turing machine sketched above will work in space $\# M (\log_2 \# M) (2^{|M|})$. From the term $M$ the Turing machine computes a string
of symbol \( w \) on the form
\[ w \equiv \frac{M_0}{p_1} : M_1 / p_2 : M_2 / \ldots / p_\ell : M_\ell \]
where \( p_1, \ldots, p_\ell \) are pointers. It is easy to see that \( \ell < \#M \). Each pointer is represented by a binary number, and thus, the number of tape cells required to represent \( w \) will be bound by \( c_0 \cdot \#M \log_2 \#M \) for some fixed \( c_0 \in \mathbb{N} \). Then the Turing machine computes \( \text{val}^V_{n+1}(M) \) by using registers. The greatest number stored in a register during the computation is bounded by \( |\xi|_{n+1} \) where \( \xi \) is a type of level \( k \). By the lemmas 1 and 2 there exist a polynomial \( p \) and a constant \( c_1 \in \mathbb{N} \) such that \( |\xi|_{n+1} \leq 2^p(n) \leq 2^{c_1|n|} \). The Turing machine represents the numbers in the registers binary, and hence the number of tape cells required for one register is bounded by \( 2^{c_2|n|} \) for some constant \( c_2 \in \mathbb{N} \). It is easy to see that the number of registers required is bounded by the length of \( w \). This entails that there exists a fixed \( c \in \mathbb{N} \) such that the Turing machine works in space \( \#M (\log_2 \#M)(2^{|n|}) \).

We will now argue that the Turing machine sketched above will work in time \( p(\#M)(2^{c_2|n|}) \) for some fixed polynomial \( p \) and constant \( c \in \mathbb{N} \). The Turing machine will a certain number of times compute the function \( \text{val}^V_{n+1}(S) \) for some subterm \( S \) and store the result away in a register. We have argued that the number of tape cells needed for one register is bounded by \( 2^{c_2|n|} \) for some constant \( c_2 \in \mathbb{N} \). Thus, the number of steps required to compute a value stored in a register will be bounded by \( 2^{c_2|n|} \) for some constant \( c_3 \in \mathbb{N} \). It is easy to see that the number of times the Turing machine computes the function \( \text{val}^V_{n+1} \) is bounded by \( p_0(\#M) \) for some polynomial \( p_0 \). Hence the number of steps in the whole computation will be bounded by \( p_1(p_0(\#M)(2^{c_2|n|})) \) for some polynomial \( p_1 \). Finally, observe that there exist a polynomial \( p \) and a constant \( c \in \mathbb{N} \) such that \( p_1(p_0(\#M)(2^{c_2|n|})) \leq p(\#M)(2^{c_2|n|}) \).

**Lemma 18.** Let \( M : 0 \) be a closed \( T \)-term on \( \beta \)-normalform. Then, we have \( \text{lev}(M) = \text{rk}(M) + 1 \).

**Proof.** Obviously, it cannot be the case that \( \text{lev}(M) < \text{rk}(M) + 1 \). Suppose that \( \text{lev}(M) > \text{rk}(M) + 1 \). Then there exists a subterm \( N : \rho \to \tau \) of \( M \) such that \( \text{lev}(N) = \text{lev}(M) = \text{dg}(\rho \to \tau) > \text{rk}(M) + 1 \). First we note that \( N \) cannot be a variable. (Because \( M \) is a closed term, and if \( N \) were a variable, we would have \( \text{lev}(M) > \text{dg}(\rho \to \tau) \).) Thus, \( N \) has the form \( \lambda x^\rho . P^\tau \), and since \( M \) has type \( 0 \) there will be a subterm of \( M \) on the form \( \lambda x^\rho . P^\tau Q^\rho \). This contradicts that \( M \) is on \( \beta \)-normalform.

**Lemma 19.** Let \( M : 0 \) be a closed \( \lambda^\Pi \)-term of level \( k + 1 \). There exists a \( \lambda^\Pi \)-term \( N \) of level \( k \) such that \( M \supset N \text{ and } \#N \leq 2^c \#M \) for some fixed \( c \in \mathbb{N} \). Moreover, the term \( N \) can be computed from \( M \) in time \( 2^c \#M \) for some fixed \( c \in \mathbb{N} \).

**Proof.** Adapt one of the standard proofs found in the literature for eliminating “cuts” in the typed \( \lambda \)-calculus. It should be straightforward to generalize such a proof to the \( \lambda^\Pi \)-calculus. See e.g. Beckmann [3] for more details.
Lemma 20. Let \( M : 0 \to 0 \) be a closed \( T^- \)-term. Then, there exists a \( \lambda \)[\( \gamma \)]-term \( N \) such that

1. \( \text{lev}(N) = \text{rk}(M) + 1 \)
2. \( M \pi = \pi \iff N = \pi \iff \text{val}_{n+1}(N) = m \)
3. and for any \( \lambda \)[\( \gamma \)]-term \( N' \) such that \( N \triangleright N' \) (that is \( N \) reduces to \( N' \)) we also have \( \text{val}_{n+1}(N) = m \).
4. \( N \) can be generated on a Turing machine's tape in time \( 2^n|n| \) where \( c \) is some fixed number.

Proof. For each \( n \in \mathbb{N} \) we define a mapping \( \Gamma_n \) of the \( T \)-terms into the \( \lambda \)[\( \gamma \)]-terms. \( \Gamma_n(x) = x \) if \( x \) is a variable; \( \Gamma_n(0) = 0 \); \( \Gamma_n(1) = s0 \); \( \Gamma_n(sM) = s\Gamma_n(M) \); \( \Gamma_n(\lambda x M) = \lambda x \Gamma_n(M) \); \( \Gamma_n((MN)) = (\Gamma_n(M)\Gamma_n(N)) \); \( \Gamma_n((M, N)) = (\Gamma_n(M), \Gamma_n(N)) \); \( \Gamma_n(\text{fst} M) = \text{fst} \Gamma_n(M) \); \( \Gamma_n(\text{snd} M) = \text{snd} \Gamma_n(M) \); and (the interesting case)

\[
\Gamma_n(\text{R}_n NFG) = \gamma(n, \Gamma_n(N), \Gamma_n(F), \Gamma_n(G))
\]

where \( \gamma(0, N, F, G) = G \) and

\[
\gamma(n + 1, N, F, G) = [n + 1, N, F \pi, \lambda X^\omega \cdot X^\omega]_\sigma \gamma(n, N, F, G)
\]

Note that for any term \( P \) there exists a polynomial \( p \) such that \( \#\gamma(n)(P) \leq p(n) \) (*).

Let \( M \) be the term given in the lemma, that is, \( M \) is some closed \( T^- \)-term of type \( 0 \to 0 \). Then \( M \pi \) is a closed term of type \( 0 \). Use \( \beta \)-reductions only on the term \( M \pi \) and obtain a term \( M' \) on \( \beta \)-normal form such that \( M' = M \pi \). By Lemma 18 we have \( \text{lev}(M') = \text{rk}(M) + 1 \). Let \( N \) be the term \( \Gamma_n(M') \). It is fairly easy to see that (i), (ii) and (iii) hold. Further, it is easy to see that the length of \( M' \) is bounded by a polynomial in \( n \), and then by (*), the length of \( N \) is also bounded by a polynomial in \( n \). This implies that there exists a fixed \( c_0 \in \mathbb{N} \) such that the length of \( N \) is bound by \( 2^n|n| \). A Turing machine can generate \( N \) straightforwardly, and thus, there will be a fixed \( c \in \mathbb{N} \) such that \( N \) can be generated in time \( 2^n|n| \). Hence, (iv) holds.

\[ \square \]

Theorem 6. \( \mathcal{J}^2i \subseteq \text{SPACE } 2^i \).

Proof. 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 \( M : 0 \to 0 \) be a closed \( T^- \)-term of rank \( 2k + 2 \). We will prove that the number \( m \) such that \( M \pi = \pi \) can be computed in space \( 2^n|n| \) and hence in space \( 2^n|n| \) for some fixed \( c \in \mathbb{N} \).

The Turing machine computing the number \( m \) will first generate a \( \lambda \)[\( \gamma \)]-term \( N \) of level \( 2k + 3 \) such that \( \text{val}_{n+1}(N) = m \). By Lemma 20 the term \( N \) can be generated in time \( 2^n|n| \) where \( c_0 \) is a fixed number. Thereafter, the Turing machine generates a \( \lambda \)[\( \gamma \)]-term \( P \) of level \( k + 3 \) such that \( P = N \). By Lemma 19, the term \( P \) can be generated in time \( 2^c|N| \) for some fixed \( c \in \mathbb{N} \), and thus in time
This entails that \( \#P \leq 2^{[c_0 + c_1][n]} \) (*). Finally, the Turing machine computes the value \( \text{val}_{n+1}(P) \). By Lemma 20 and Lemma 19 the computed value will be the number \( m \) such that \( \overline{M\overline{m}} = \overline{m} \). Lemma 17 says that \( \text{val}_{n+1}(P) \) can be computed in space \( \#P[\log_2 \#P] 2^{[c_0][n]} \) for some fixed \( c_0 \in \mathbb{N} \). We have

\[
\#P[\log_2 \#P] 2^{[c_0][n]} \leq 2^{[c_0 + c_1][n]} (\log_2 2^{[c_0 + c_1][n]}) 2^{[c_0][n]} \leq 2^{[c][n]}
\]

for a sufficiently large \( c \in \mathbb{N} \), and thus the Turing machine works space \( 2^{[c][n]} \).

Case \( i = 0 \). Let \( M : 0 \to 0 \) be a closed \( T \)-term of rank 0. Let \( x : 0 \) be a variable not occurring in \( M \), and let \( P \) be \( Mx \) on \( \beta \)-normalform. A Turing machine can compute \( \text{val}_{n+1}(P[x := \overline{m}]) \) using \( k_0 \) registers holding numbers of type 0. (A register is noting 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 \( 0_{n+1} = \{0, \ldots, n\} \). Thus, the number \( m \) such that \( \overline{m} = M\overline{m} \) can be computed in space \( k|n| \) for some fixed \( k \in \mathbb{N} \).

This proves that \( \mathcal{F}^{2^i} \subseteq \text{SPACE}\ 2^{2^{2^i}} \).

**Theorem 7.** \( \mathcal{F}^{2^{i+1}} \subseteq \text{TIME}\ 2^{2^{2^{i+1}}} \).

**Proof.** The proof splits into the cases \( i = 0 \) and \( i > 0 \). The two cases are very similar, but not uniform, and we have to treat them separately.

Case \( i > 0 \). Then \( i = k + 1 \) and \( 2i + 1 = 2k + 3 \) for some \( k \in \mathbb{N} \). Let \( M : 0 \to 0 \) be a closed \( T \)-term of rank \( 2k + 3 \). We will prove that the number \( m \) such that \( \overline{M\overline{m}} = \overline{m} \) can be computed in time \( 2^{c_0}[n] \) (and hence in time \( 2^{c_0}[n] \)) for some fixed \( c_0 \in \mathbb{N} \). The Turing machine computing the number \( m \) will first generate a \( \lambda \)-term \( N \) of level \( 2k + 4 \) such that \( \text{val}_{n+1}(N) = m \). By Lemma 20 the term \( N \) can be generated in time \( 2^{c_0}[n] \) where \( c_0 \) is a fixed number. Thereafter, the Turing machine generates a \( \lambda \)-term \( P \) of level \( k + 3 \) such that \( P = N \). By Lemma 19, the term \( P \) can be generated in time \( 2^{c_1}[N] \) for some fixed \( c_1 \in \mathbb{N} \), and thus in time \( 2^{[c_0 + c_1][n]} \). This entails that \( \#P \leq 2^{[c_0 + c_1][n]} \) (*). Finally, the Turing machine computes the value \( \text{val}_{n+1}(P) \). By Lemma 20 and Lemma 19 the computed value will be the number \( m \) such that \( \overline{M\overline{m}} = \overline{m} \). Lemma 17 says that \( \text{val}_{n+1}(P) \) can be computed in time \( p(\#P) 2^{[c_0][n]} \) for some fixed polynomial \( p \) and constant \( c_3 \in \mathbb{N} \). We have

\[
p(\#P) 2^{[c_0][n]} \leq p(2^{[c_0 + c_1][n]}) 2^{[c_3][n]} \leq 2^{[c][n]}
\]

for some sufficiently large \( c \in \mathbb{N} \), and thus the Turing machine works in time \( 2^{[c][n]} \).

Case \( i = 0 \). Proceed as in the case \( i > 0 \), but do not reduce the level of the intermediate term \( N \). Compute \( \text{val}_{n+1}(N) \) straightaway and use Lemma 17 to argue that the number of steps in computation will bounded by \( 2^{[c][n]} \) for some fixed \( c \in \mathbb{N} \). (If this case were uniform to the case \( i > 0 \), the level of the intermediate term \( N \) should have been reduced by 1 before the Turing machine embarked on the algorithm given in the proof of Lemma 17.)

This proves that \( \mathcal{F}^{2^{i+1}} \subseteq \text{TIME}\ 2^{2^{2^{i+1}}} \).
Acknowledgments. The authors would like to thank G. Mathias Barra for helpful remarks on various drafts of this paper.
References