Control Structures in Programs
and
Computational Complexity

Habilitationsschrift
zur Erlangung des akademischen Grades Dr. rer. nat. habil.
vorgelegt der
Fakultät für Informatik und Automatisierung
an der Technischen Universität Ilmenau

von
Dr. rer. nat. Karl–Heinz Niggl

Referees:

Prof. Dr. Klaus Ambos-Spies
Prof. Dr.(USA) Martin Dietzfelbinger
Prof. Dr. Neil Jones

Day of Defence: 2nd May 2002

In the present version, misprints are corrected in accordance with the referee’s comments, and some proofs (Theorem 3.5.5 and Lemma 4.5.7) are partly rearranged or streamlined.
Abstract

This thesis is concerned with analysing the impact of nesting (restricted) control structures in programs, such as primitive recursion or loop statements, on the running time or computational complexity.

The method obtained gives insight as to why some nesting of control structures may cause a blow up in computational complexity, while others do not. The method is demonstrated for three types of programming languages. Programs of the first type are given as lambda terms over ground-type variables enriched with constants for primitive recursion or recursion on notation. A second is concerned with ordinary loop programs and stack programs, that is, loop programs with stacks over an arbitrary but fixed alphabet, supporting a suitable loop concept over stacks. Programs of the third type are given as terms in the simply typed lambda calculus enriched with constants for recursion on notation in all finite types.

As for the first kind of programs, each program \( t \) is uniformly assigned a measure \( \mu(t) \), being a natural number computable from the syntax of \( t \). For the case of primitive recursion, it is shown that programs of \( \mu \)-measure \( n + 1 \) compute exactly the functions in Grzegorczyk level \( n + 2 \). In particular, programs of \( \mu \)-measure 1 compute exactly the functions in \( \text{FLINSPACE} \), the class of functions computable in binary on a Turing machine in linear space. The same hierarchy of classes is obtained when primitive recursion is replaced with recursion on notation, except that programs of \( \mu \)-measure 1 compute precisely the functions in \( \text{FPTIME} \), the class of the functions computable on a Turing machine in time polynomial in the size of the input.

Another form of measure \( \mu \) is obtained for the second kind of programs. It is shown that stack programs of \( \mu \)-measure \( n \) compute exactly the functions computable by a Turing machine in time bounded by a function in Grzegorczyk level \( n + 2 \). In particular, stack programs of \( \mu \)-measure 0 compute precisely the \( \text{FPTIME} \) functions. Furthermore, loop programs of \( \mu \)-measure \( n \) compute exactly the functions in Grzegorczyk level \( n + 2 \). In particular, loop programs of \( \mu \)-measure 0 compute precisely the \( \text{FLINSPACE} \) functions.

As for the third kind of programs, building on the insight gained so far, it is shown how to restrict recursion on notation in all finite types so as to characterise polynomial-time computability. The restrictions are obtained by using a ramified type structure, and by adding linear concepts to the lambda calculus. This gives rise to a functional programming language \( \text{RA} \) supporting recursion on notation in all finite types. It is shown that \( \text{RA} \) programs compute exactly the \( \text{FPTIME} \) functions.
1. Introduction

In this thesis the objects under consideration are programs built from basic operations, such as “increase a number by one” or “add ground data to the current storage”, by restricted control structures, such as sequencing and for-do loops in imperative programming languages, or equivalently, composition and primitive recursion or recursion on notation\(^1\), or else by lambda abstraction, application and higher-type recursion in functional programming languages.

This work is concerned with the following simple question: *From the syntax of a given program, can one conclude its running time or computational complexity?*

For example, *can one extract information out of the syntax of programs so as to separate programs which run in polynomial time (in the size of the input) from those which may not?*

And if so, is there a general rationale behind so as to separate programs which run in polynomial time from programs of (iterated) exponential running time, and the latter from programs which run in super-exponential time, and so forth?

 Obviously, it is the nesting of for-do statements or recursion that can lead to a blow up in running time. However, some nesting of control structures do not cause such a blow up, while others do. In this thesis, syntactical criteria are given that separate nesting of such control structures which do not cause a blow up in computational complexity from those which might.

In case of imperative programming languages and ground type recursion, this gives rise to a measure \(\mu\) that assigns in a purely syntactic fashion to each program \(P\) in the language a natural number \(\mu(P)\). It is shown that the measure \(\mu\) on loop programs and lambda terms over ground type variables and primitive recursion characterises the Grzegorczyk hierarchy at and above the linear-space level. Furthermore, the measure \(\mu\) on stack programs and lambda terms over ground type variables and recursion on notation characterises Turing machine computability with running time bounded by functions in the Grzegorczyk hierarchy at and above the linear-space level.

In particular, stack programs of \(\mu\)-measure 0 compute exactly the polynomial-time computable functions, and loop programs of \(\mu\)-measure 0 compute precisely the linear-space computable functions.

These results are put in perspective by showing that there are limitations intrinsic to any such method: No computable measure on stack programs can identify exactly those programs which run in polynomial time.

Building on the insights as to how super-polynomial running time comes about, it is shown how to restrict recursion on notation in all finite types so as to characterise polynomial-time computability. The restrictions are obtained by using a ramified type structure, and by adding linear concepts to the lambda calculus. This gives rise to a functional programming language RA, supporting recursion on notation in all finite types. It is shown that RA programs compute exactly the polynomial-time computable functions.

To explain the main ideas behind the measure \(\mu\), we first focus on the usual schemes for defining primitive recursive functions. Building on the definition of addition by primitive recursion from successor, that is,

\[
\begin{align*}
\text{add}(0, y) &= y \\
\text{add}(x + 1, y) &= \text{add}(x, y) + 1
\end{align*}
\]

consider the following definitions of multiplication and exponentiation:

\[
\begin{align*}
\text{mult}(0, y) &= 0 \\
\text{mult}(x + 1, y) &= \text{add}(y, \text{mult}(x, y)) \\
\text{exp}(0) &= 1 \\
\text{exp}(x + 1) &= \text{add}(\text{exp}(x), \text{exp}(x))
\end{align*}
\]

\(^1\)Recursion on notation is recursion on the natural numbers, but in binary representation (cf. Background 2.4).
Both functions are defined by one recursion from addition. From outside we know, of course, that \textit{mult} runs in polynomial-time, while \textit{exp} must run in super-polynomial time. But given these recursion equations only, how does this blow up in computational complexity come about?

The gist of the matter lies in different “control” of the input positions: In either case, “input position” \( x \) controls a recursion, but while \textit{mult} passes its computed values to the “input position” \( y \) of \textit{add}, which has no control over any other recursion, each computed value of \textit{exp} controls the recursion \textit{add} is defined by.

Thus, in order to follow up the “control” of each input position in a computation, to each definition of a primitive recursive function \( f(x_1, \ldots, x_l) \), one assigns ranks \( n_1, \ldots, n_l \) to all “input positions” \( x_1, \ldots, x_l \), the intuition being that "input position" \( x_i \) controls \( n_i \) “top recursions”

“Top recursions” are the only form of recursion that may cause a blow up in computational complexity, while all other forms, called “side recursions”, do not. It is understood that the maximum of all these ranks defines the “computational complexity of \( f \)".

To see the mechanism in the critical case, suppose that \( f \) is defined by primitive recursion from \( g, h \), say
\[
\begin{align*}
g(x_1, \ldots, x_l) & \text{ has ranks } m_1, \ldots, m_l \\
h(x_1, \ldots, x_l, u, v) & \text{ has ranks } n_1, \ldots, n_l, p, q
\end{align*}
\]
Then each input position \( x_i \) of \( g, h \), being a “parameter position” with respect to the recursion \( f \) is defined by, can “contribute” its ranks to the rank of input position \( x_i \) of \( f \), and input position \( y \), which controls the recursion, essentially increments by one the rank \( q \) of the “critical input position” \( v \) in the recursion, that is,
\[
f(x_1, \ldots, x_l, y) \text{ has ranks } \max(m_1, n_1, q), \ldots, \max(m_l, n_l, q), \max(p, 1 + q).
\]
To provide the information that each parameter \( x_i \) participates in a “recursion at rank \( q \)”, the rank of each \( x_i \) is at least \( q \). This bookkeeping of the “control” in \( f \) allows one to distinguish two forms of recursion:
\[
f \text{ is a } \begin{cases} 
top recursion & \text{ if } q \geq \max\{m_1, n_1, \ldots, m_l, n_l, p\} \\
side recursion & \text{ else}
\end{cases}
\]
Now, adding that each input position of an “initial function”, such as the successor function \( S(x) \), will have rank 0, one easily verifies for the examples above that the definition of \textit{add} has ranks 1, 0 and, therefore, the definition of \textit{mult} has ranks 1, 1, since it is a side recursion, while the definition of \textit{exp} has rank 2, because it is a top recursion.

In fact, the primitive recursive programs of \( \mu \)-measure 1, allowing \textit{one} top recursion (like \textit{add}) and \textit{any number} of side recursions (like \textit{mult}), compute exactly the functions computable in unary by a Turing machine in polynomial time – the functions at Grzegorczyk level 2. This perfectly generalises to all levels of the Grzegorczyk hierarchy.

The definition of the measure \( \mu \) is, however, complicated, in that it accounts for “redundant input” positions in order to assign ranks as low as possible. So for a complete picture (cf. chapter 2), each rank assigned to an input position is either a natural number, as assumed above for simplicity, or else it is \( \iota \) whenever this input position is (observationally) redundant.

The requirement of recognising redundant input positions is not a futile or merely purist-like venture in an altogether imperfect situation, but rather helps to identify at each level as many programs as possible, in
the first place, and allows one to organise proofs in a uniform way, be it primitive recursion or recursion on notation, in the second place.

In case of imperative programs, the somewhat involved bookkeeping used to follow up the “control” in primitive recursive programs can be dispensed with in favour of a conceptually simple graph-theoretical analysis of “control”. The main ideas behind such analysis are explained in terms of stack programs.

Stack programs operate with stacks $X, Y, Z, \ldots$ over a fixed but arbitrary alphabet. Such programs are built up from the usual instructions $\text{push}(a, X), \text{pop}(X), \text{nil}(X)$ by sequencing, conditional statements $\text{if top}(X) \equiv a [Q]$ and loop statements $\text{foreach } X [Q]$. The operational semantics of stack programs is standard; in particular, loop statements are executed call-by-value, allowing one to inspect every symbol on the control stack while preserving its contents.

To see this, for loop statements $\text{foreach } X [P]$, we require that no instruction $\text{push}(a, X), \text{pop}(X)$ or $\text{nil}(X)$ occurs in the body $P$. Thus, the control stack $X$ can not be altered during an execution of the loop. To provide access to each symbol on the control stack $X$ during an execution of the loop, first a local copy $U$ of $X$ is allocated, and $P$ is altered to $P'$ by simultaneously replacing each “free occurrence” of $X$ in $P$ (appearing as if $\text{top}(X) \equiv a [Q]$) with $U$. Then the sequence

$$P'; \text{pop}(U); \ldots; P'; \text{pop}(U) \quad (|X| \text{ times})$$

is executed. As in lambda calculus, an occurrence of $X$ in $P$ is free if it does not appear in the body $Q$ of a subprogram $\text{foreach } X [Q]$ of $P$.

As with recursion, some nestings of loops do not cause a blow up in running time, while others do. A case in point is the following program $P_1$ which, without question, runs in polynomial time:

$$P_1 \equiv \text{foreach } X_1 […] \text{foreach } X_l [\text{push}(a, Y)]$$

For if words $v_1, \ldots, v_l, w$ are stored in $X_1, \ldots, X_l, Y$ respectively before $P_1$ is executed, then $Y$ holds the word $a^{2^{|w|}} \cdot v_1 \cdots v_l$ after the execution of $P_1$. What can be read from this example is that “directly nesting loops” does not lead to a blow up in running time. In fact, each $X_i$ controls $Y$ in the sense that $Y$ is pushed inside the body of the loop governed by $X_i$. But no “computed value” in the body of a loop of program $P_1$ controls another loop. This could only happen if some body of a loop were a sequence. But even then, a blow up in running time can only occur if the body of a loop, being a sequence, has a control circle, in that some $Z$ controls some $Y$, and eventually $Z$ controls $Y$. To see this, consider the following programs:

$$P_2 \equiv \text{nil}(Y); \text{push}(a, Y); \text{nil}(Z); \text{push}(a, Z); \text{foreach } X [\text{nil}(Z); \text{foreach } Y [\text{push}(a, Z); \text{push}(a, Z)]; \text{nil}(Y); \text{foreach } Z [\text{push}(a, Y)]]$$

$$P_3 \equiv \text{nil}(Y); \text{push}(a, Y); \text{nil}(Z); \text{foreach } X [\text{foreach } Y [\text{push}(a, Z); \text{push}(a, Z)]; \text{push}(a, Y)]$$

Both programs have “loop nesting depth” 2, and at first glance, they look quite similar. But on inspection we see that $P_2$ runs in exponential time, whereas $P_3$ has polynomial running time. For if $w$ is initially stored in $X$, then $Z$ holds the word $a^{2^{|w|}}$ after $P_2$ is executed, while $a^{|w|+(|w|+1)}$ is stored in $Z$ after the execution of $P_3$. The reason is simply that in the body of the outermost loop of $P_2$, first $Y$ controls $Z$, and then $Z$ controls $Y$, in the sense above. In contrast, there is no such circle in $P_3$.

In fact, it turns out that stack programs with only circle-free loop bodies compute exactly the polynomial-time computable functions.
Again, this perfectly generalises to all levels of the Grzegorczyk hierarchy. The $\mu$-measure of a primitive instruction is 0, the $\mu$-measure of a sequence is the maximal $\mu$-measure of its components, and decisively,
\[
\mu(\text{foreach } X [Q]) := \begin{cases} 
\mu(Q) + 1 & \text{if } Q \text{ is a sequence with a top circle} \\
\mu(Q) & \text{else}
\end{cases}
\]
where a sequence $Q_1; \ldots; Q_l$ has a top circle if there exists a component $Q_i$ of maximal $\mu$-measure such that some $Y$ controls some $Z$ in $Q_i$, and $Z$ controls $Y$ in $Q_1; \ldots; Q_{i-1}; Q_{i+1}; \ldots; Q_l$.

One easily verifies for the examples above that $P_1, P_3$ are of $\mu$-measure 0, while $P_2$ is of $\mu$-measure 1.

Given the above characterisations of the polynomial-time computable functions by either stack programs with only circle-free loop bodies, or by recursion on notation of $\mu$-measure 1, how can one benefit from it in order to characterise the polynomial-time computable functions by “higher type recursion on notation”?

Recall that in terms of recursion on notation, programs of $\mu$-measure 1 permit one top recursion, and as many side recursions as desired. In other words, the rationale behind such programs is that computed values in a recursion must not control other recursions. Undoubtedly, this will remain valid for “higher type recursion on notation”, too. But in the presence of “higher type recursion”, new phenomena and problems immediately arise which must also be kept under control.

“Higher type recursion” has long been viewed as a powerful scheme unsuitable for describing small complexity classes such as polynomial time. In fact, untamed recursion in all finite types computes exactly the $\epsilon_0$-recursive functions (cf. Schwichtenberg [57] for a modern approach). Furthermore, in the presence of “higher-type inputs”, computed functionals in higher-type recursions can be “nested” such that by a single higher-type recursion one can define functions of exponential growth.

To demonstrate this, we focus on higher-type recursion in terms of the natural numbers in unary representation. Recursion in all finite types was introduced by Hilbert [26] and later became known as the essential part of Gödel’s system $T$ [21]. This recursion can be thought of as recursion over the natural numbers, but with the characteristic that the computed values need not be numbers, but can be any “functional”, a mapping that takes other functionals as arguments and returns numbers or again functionals.

To make this notion precise, Hilbert introduced the set of (simple) types, inductively defined as follows: $\iota$ is a type called “ground type”, and if $\sigma, \rho$ are types, then so is the “function type” $(\sigma \to \rho)$. Each type $\sigma$ has associated with it a set $H_\sigma$ of functionals of type $\sigma$ inductively defined by:
\[
H_\iota := \mathbb{N} \\
H_\sigma \to \rho := \text{the set of all mappings } F : H_\sigma \to H_\rho
\]
For readability, given arguments $X_1, \ldots, X_k$, one often writes $F(X_1, \ldots, X_k)$ for $F(X_1) \ldots (X_k)$.

Now a functional $F$ of type $\iota \to \sigma$ is defined by recursion in type $\sigma$ from functionals $G, H$ of types $\iota, (\iota \to \sigma \to \sigma)$ respectively, denoted by $R_\sigma(G, H)$, if $F$ satisfies the following “recursion equations”:
\[
F(0) = G \\
F(x + 1) = H(x, F(x))
\]
It is understood that $G, H$ are defined beforehand where, apart from recursion in any type, one can use explicit definitions of the form $F'(X_1, \ldots, X_k) = \text{RHS}$, where the right-hand side is any functional expression.

\[2\]More on these concepts and an introduction to (higher-type) recursion on notation can be found in Background 2.1 and 2.4.
built up from the variables $X_1, \ldots, X_k$ and constants by previously defined functional symbols distinct from $F'$, including initial functional symbols, such as $S$ for the successor function.

Now, observe that by a single higher-type recursion one can define the function $e(x, y) = 2^x + y$. One simply writes the equations $e(0, y) = 1 + y$ and $e(x + 1, y) = e(x, e(x, y))$ as recursion in type $\iota \rightarrow \iota$; that is, one obtains $e = \mathcal{R}_{\iota \rightarrow \iota}(S, H)$ with $H$ (explicitly) defined by

$$H(x, f, y) = f(f(y)).$$

So we see that exponential growth results from higher-type recursion by iterating or nesting the computed functional, $e(x, y)$ above, a constant number of times. However, if the number of iterations is linked to an input of the functional defined, then one can define functions of non-primitive recursive growth rate. A case in point is the following binary version $A$ of the well-known Ackermann function, defined by

$$A(0, y) = 1 + y$$
$$A(x + 1, y) = A(x)^{(y+1)}(y)$$

where for unary $f$, $f^{(k)}$ denotes the $k$th iterate of $f$, that is, $f^{(0)}(x) = x$ and $f^{(k+1)}(x) = f(f^{(k)}(x))$. First observe that the iteration functional $I$ satisfying $I(x, f, y) = f^{(x)}(y)$ can be defined by recursion in type $(\iota \rightarrow \iota) \rightarrow \iota \rightarrow \iota$, since $I$ has the following “recursion equations”:

$$I(0, f, y) = y$$
$$I(x + 1, f, y) = f(I(x, f, y))$$

Thus, we obtain $A = \mathcal{R}_{\iota \rightarrow \iota}(S, H)$ for the functional $H$ (explicitly) defined by $H(x, f, y) = I(y + 1, f, y)$.

What can be read from this example is that non-primitive recursive growth rate results from higher-type recursion by applying computed functionals, $A(x)$ above, of an outer recursion to computed functionals, i.e. $I(x, A(x), y)$, of an inner recursion.

Thus, to set up a calculus in which the use of higher-type recursion (on notation) and explicit definition is controlled in a purely syntactic fashion so as to characterise polynomial-time computability, we introduce at the same time both ramification of the type structure and linearity conditions for the use of computed functionals in recursions.

In terms of ramification, the idea is as simple as this: Some input positions of functionals defined in the calculus are labelled with a mark $!$, that is, we enrich the type structure with the formation of types $!\sigma$, called complete types; all other types are called incomplete. Intuitively,

- Complete higher type can be used in any non-linear way
- Incomplete higher type can be used in a certain linear way only
- Type $!\iota$ control at most one top recursion
- Type $\iota$ control no recursion.

By ramification, one can rule out non-primitive recursive growth rate and generalise the concept that computed values in a recursion must not control other recursions with the following two requirements:

(i) Recursion in type $\sigma$, say $F = \mathcal{R}_{\sigma}(G, H)$, is only admitted for $!$-free types $\sigma$, called safe types, and for $H$ of complete types $!(\iota \rightarrow \sigma \rightarrow \sigma)$; $F$ itself receives type $!\iota \rightarrow \sigma$.

(ii) Functional expressions, like the right-hand sides of explicit definitions, can only receive a complete type if they contain no incomplete variables.
Observe that (i),(ii) do not rule out the nesting of computed functionals in recursions, such as in the definition of the function $e$ above. This is precluded by another concept, called affinability, which is central to the calculus and expresses the linearity constraints for higher type incomplete input positions:

(iii) An input position $X_i$ of a functional $F(X_1, \ldots, X_k)$ defined by explicit definition is of incomplete type if it is affinable in that definition, that is, the right-hand side either has at most one occurrence of $X_i$, or else it contains a functional expression $a$ of type $\iota$ such that $a$ has a single occurrence of $X_i$, and every occurrence of $X_i$ in the right-hand side is in an occurrence of $a$, which is to say,

$$F(X_1, \ldots, X_k) = \ldots a \ldots a \ldots a \ldots$$

such that the occurrences of $X_i$ are separated by one and the same ground type context $a$.

These restrictions give rise to a system $\text{RA}$ of ramified affinable terms which allows one top recursion in any type, and as many side recursions in any type as desired. In fact, $\text{RA}$ programs compute exactly the polynomial-time computable functions.

Much of this research has been motivated by outstanding work in logic (cf. [61], [20], [2], [55], [56], [8], [17], [9], [10]) initiated by Kreisel [30], resulting in a new method, “proofs-as-programs”, for the development of correct software. It nurtures the vision that a future “programmer” can develop in an interactive proof system environment both the underlying data types the program in search of is operating on, and a formal proof, $d$ say, of the specification of the program. Such specifications formally express input/output behaviour in the form “for every input $\vec{x}$ there exists an output $y$ such that $\text{spec}(\vec{x}, y)$ holds”, where $\text{spec}(\vec{x}, y)$ is a quantifier-free formula. The method then allows one to automatically extract from the internally represented proof $d$ a program $\text{ep}(d)$ that is correct with respect to the formalised specification, that is, $\text{spec}(\vec{x}, \text{ep}(d)(\vec{x}))$ is true for all input $\vec{x}$. Notably, the method works for both intuitionistic and classical proofs. However, to make this method fruitful and applicable in industry, where safety has high priority, one must resolve a drawback intrinsic to the method: extracted programs make use of recursion in all finite types; in other words, they are objects in one or another version of system $T$, and until recently, there was no method available to analyse the running time of such programs.

Thus, the present work can be considered a contribution to resolving this drawback, for it proposes a purely syntactical method for analysing the running time of extracted programs, and as pointed out above, applies neatly to imperative programs with the flavour of realistic imperative languages, too. For these reasons, it is natural to understand first how this can be achieved for lambda terms over ground type variables enriched with ground type recursion.

The result that programs of $\mu$-measure $n$, $n \geq 1$, given as lambda terms over ground type variables and primitive recursion, compute exactly the functions at Grzegorczyk level $n + 1$ is an improvement on other characterisations of the Grzegorczyk hierarchy at and above level 3 by Schwichtenberg [54], Müller [41] and Parsons [49], which are all based on the “nesting depth” of primitive recursions (cf. Heinermann [25]). Thus, the improvement is that programs of $\mu$-measure 1 compute exactly the functions computable in binary on a Turing machine in linear space.

The result that one obtains the same series of classes except with the polynomial-time computable functions at first level, when primitive recursion is replaced with recursion on notation, generalises the result of Bellantoni and Cook [5] (cf. Background 2.8) to all levels of the Grzegorczyk hierarchy. The measure $\mu$, however, is an improvement on the result of Bellantoni and Cook, for various reasons. Firstly, it operates on pure lambda terms without any kind of bookkeeping, such as that used to set up the function algebra $BC$.
in [5]. Thus, programs can be written in the usual way. Secondly, each level of $\mu$ is naturally closed under composition, unlike the function algebra $BC$. Thirdly, the measure $\mu$ accounts for redundant input positions such that altogether significantly more algorithms can be identified at each level than by any other known measure.

The measure $\mu$ is similar in spirit to the ranking $\rho$ in [6], which operates on the usual definition schemes of the primitive recursive functions. The ranking $\rho$ gives elegant characterisations of the same complexity classes discussed here. Compared to the measure $\mu$, it is conceptually simpler, one reason being that it does not account for redundant input positions. As a result, however, apart from the functions zero and successor, $\rho$ does require further initial functions. Furthermore, while the proof that the polynomial-time computable functions can be simulated by programs of $\rho$-measure 1 relies on the, in parts hard going, proof given in [5], the measure $\mu$ allows one to give an intuitive and lucid proof, being just a binary version of the “simulation trick” (developed in [46] and used in [6]) in terms of primitive recursion.

The present work on the measure $\mu$ operating on lambda terms over ground type variables and ground type recursion therefore builds on recent work on ramified analysis of recursion by Bellantoni and Niggl [6], and Niggl [46]. It can be considered a self-contained and modified version of the latter.

The result that loop programs of $\mu$-measure $n$ compute exactly the functions at Grzegorczyk level $n + 2$ improves the result of Meyer and Ritchie [40], where the nesting depth of “loop programs” is related to the Grzegorczyk hierarchy at and above level 3.

The result that stack programs of $\mu$-measure $n$ compute exactly the functions computable on a Turing machine in time bounded by a function at Grzegorczyk level $n + 2$ is of interest for various reasons. Firstly, stack programs of $\mu$-measure 0 compute precisely the polynomial-time computable functions. Secondly, the measure $\mu$ is conceptually simple, and it operates on programs written in a simple programming language which nonetheless has the flavour of realistic imperative languages, with the promise that it might be extended to actual programming languages. Thirdly, one can argue that the measure $\mu$ is likely to give the minimal complexity for a great deal of natural algorithms, and furthermore, it admits significantly more algorithms in each complexity class than any other known complexity measure on loop programs, such as the “nesting depth”. Altogether, the measure $\mu$ can help to ground the concepts of computational complexity by providing a reference point other than the original resource-based concepts.

The work on loop and stack programs is joint work with Lars Kristiansen, the paper version [31] of which is to appear in a forthcoming special issue of TCS on “Implicit Computational Complexity”.

Various ramification concepts as initiated by Simmons [59], Leivant [32, 33, 34], Bellantoni and Cook [5] have led to resource-free, purely functional characterisations of many complexity classes, such as the polynomial-time computable functions [5, 36, 37], the linear-space computable functions [3, 34, 43], $NC^1$ and polylog space [11], NP and the polynomial-time hierarchy [4], the Kalmár-elementary functions [47], and the exponential time functions of linear growth [14], among many others. Recently, Oitavem and Bellantoni [48] characterised NC using “recursion over binary trees” and a modification of rank $\rho$ above.

As pointed out above, extensions of ramification concepts (conditions (i), (ii) above) in conjunction with a liberalised form of linearity for computed functionals in higher type recursions (condition (iii) above) led to the design of system RA. Although this system supports recursion on notation in all finite types, the result is that RA programs compute exactly the polynomial-time computable functions. This work evolved from work on higher type recursion in collaboration with S. Bellantoni and H. Schwichtenberg, and it has meanwhile been integrated in a joint publication [7].

The approach to higher-type functions taken in this work contrasts with Cook and Kapron’s well-known Basic Feasible Functions (BFF) defined by $PV^\omega$ terms [18]. There, another mode of ground type recursion
on notation with explicit size bounds is used. As a result, after normalising programs in $PV^\omega$, all higher type aspects are gone, in contrast to programs in $RA$. A further difference can be seen by the fact that system $RA$ admits the iteration functional $I_t$ satisfying $I_t(f, x, y) = f^{(|x|)}(y)$, whereas $I_t$ is not BFF.

Historically, Simmons [59] (cf. Background 2.7) was the first to give syntactic restrictions for the use of recursion in type $\mathfrak{t} \rightarrow \mathfrak{t}$ so as to characterise the primitive recursive functions. Leivant [35] generalised Simmons’ ramification concept to all finite types in order to characterise the Kalmár-elementary functions. Leivant and Marion [38] showed that another form of ramification can be used to restrict higher type recursion to PSPACE.

Hofmann [27, 28] also used ramification and linearity concepts in order to define a lambda calculus enriched with constants for recursion in all finite types. This interesting work also characterises polynomial-time computability. However, the proof methods of the two characterisations are completely different, as Hofmann uses a category-theoretic approach.

Recently, building on the results of system RA, Schwichtenberg [58] designed a proof system for which the extracted programs compute precisely the polynomial-time computable functions. Furthermore, building on system RA, in particular on the evaluation strategy of RA programs on given input, Aehlig, Johannsen, Schwichtenberg and Terwijn [1] gave a characterisation of NC by recursion on notation in all finite types, additionally using Clote’s concatenation recursion on notation (cf. [13, 15]) in order to simulate nondeterminism in the recursion.

This thesis is organised in chapters. In the subsequent Chapter 2, basic concepts are introduced and their results are reviewed, provided they are, in one way or another, relevant for the development of the research reported here. Chapter 3 presents work on the measure $\mu$ operating on lambda terms enriched with two modes of recursion: primitive recursion and recursion on notation. The results for the measure $\mu$ in terms of imperative programming languages (loop and stack programs) are developed in Chapter 4. Chapter 5 presents the characterisation of the polynomial-time computable functions by recursion in all finite types.
2. Background

In this chapter, basic concepts are introduced and fundamental results of these concepts are reviewed.

2.1. Higher type recursion and Gödel’s system $T$

Recursion in all finite types was introduced by Hilbert [26] and later [21] became known as the essential part of Gödel’s system $T$. Extending Hilbert’s finitistic point of view, Gödel reduced the consistency of arithmetic to that of system $T$.

This founded the study of higher-type functionals which has had great influence on science far beyond mathematical logic. The study of higher type functionals was essential for the development and optimisation of programming languages; particularly functional programming languages use higher type constructions. Early programming languages such as Fortran already allowed functions to be passed as parameters to subprograms. Complex (data) types are also fundamental for modern programming languages such as ML, Miranda or Haskell.

System $T$ can be thought of as simply typed lambda calculus enriched with two kinds of “constants”. One is for constructing “ground data”, i.e. “numerals” $S \ldots S0$ built from constant symbols 0 for zero and $S$ for the successor function. The other is for “recursion in all finite types”, that is, recursion over ground data but with the characteristic that the computed values need not be ground, but can be any “functional”, a mapping that takes other functionals as arguments and returns ground data or functionals.

To make this notion precise, Hilbert introduced the set of (simple) types, inductively defined by: $\iota$ is a type called “ground type”, and if $\sigma, \rho$ are types, then so is the “function type” $(\sigma \rightarrow \rho)$. By repeatedly decomposing the right-hand side of $\rightarrow$, any type $\sigma \neq \iota$ can be written uniquely as

$$\sigma = (\sigma_0 \rightarrow (\sigma_1 \rightarrow \ldots \rightarrow (\sigma_{k-1} \rightarrow \iota) \ldots))$$

often written as $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots \rightarrow \sigma_{k-1} \rightarrow \iota$, or just $\vec{\sigma} \rightarrow \iota$, with the convention of association to the right. Each type $\sigma$ has associated with it a set $H_\sigma$ of functionals of type $\sigma$ inductively defined by:

$$H_\iota := \mathbb{N}$$
$$H_{\sigma \rightarrow \rho} := \text{the set of all mappings } f : H_\sigma \rightarrow H_\rho$$

Observe that functionals of type $\sigma \rightarrow \rho \rightarrow \tau$ can be identified with mappings $f : H_\sigma \times H_\rho \rightarrow H_\tau$. Thus, the notation $(\sigma_0, \sigma_1, \ldots, \sigma_{k-1} \rightarrow \iota)$ is also used, and one may think of a functional $F \in H_\sigma$ as a mapping taking functionals $X_0, \ldots, X_{k-1}$ of types $\sigma_0, \ldots, \sigma_{k-1}$ respectively as arguments, and returning a natural number. Accordingly, one writes $F(X_0, \ldots, X_{k-1}) = y$ for $F(X_0) \ldots (X_{k-1}) = y$ when convenient.

Now a functional $F$ of type $\iota \rightarrow \sigma$ is defined by recursion in type $\sigma$ from functionals $G, H$ of types $\sigma, (\iota \rightarrow \sigma \rightarrow \sigma)$ respectively, denoted by $R_\sigma(G, H)$, if $F$ satisfies the following “recursion equations”:

$$F(0) = G$$
$$F(x + 1) = H(x, F(x))$$

For example, following the recursion equations for addition, that is,

$$\text{add}(0, y) = y$$
$$\text{add}(x + 1, y) = \text{add}(x, y) + 1$$
one obtains \( \text{add} = R_{\iota \to \iota}(G, H) \) with \( G, H \) defined by \( G(y) = y \) and \( H(x, f, y) = f(y) + 1 \). Then, following the recursion equations for multiplication, that is,

\[
\begin{align*}
\text{mult}(0, y) &= 0 \\
\text{mult}(x + 1, y) &= \text{add}(y, \text{mult}(x, y))
\end{align*}
\]

one obtains \( \text{mult} = R_{\iota \to \iota}(G, H) \) with \( G, H \) now defined by \( G(y) = 0 \) and \( H(x, f, y) = \text{add}(x, f(y)) \). In that way, by successively nesting recursions, we see that every primitive recursive function can be defined by recursion in type \( \sigma \) with level \( l(\sigma) \leq 1 \), where level \( l(\sigma) \) of \( \sigma \) is defined inductively by: \( l(\iota) := 0 \) and \( l(\sigma \rightarrow \rho) := \max\{l(\sigma), 1 + l(\rho)\} \).

Now, observe that by a single higher type recursion one can define functions of exponential growth. For example, consider the following binary version of the well-known Ackermann function defined by

\[
\begin{align*}
\text{add}(x, y) &= x + y \\
\text{mult}(x, f, y) &= f(x) \cdot y
\end{align*}
\]

as recursion in type \( \iota \to \iota \), one obtains \( e = R_{\iota \to \iota}(S, H) \) with \( S \) being the successor function and \( H \) defined by \( H(x, f, y) = f(f(y)) \). This function satisfies \( e(x, y) = 2^x + y \). So we see that exponential growth results from higher type recursion by iterating the computed functional, \( e(x, y) \) above, a constant number of times. In fact, if the number of iterations is linked to one of the inputs of the functional defined, then one can define functions of non-primitive recursive growth rate. For example, consider the following binary version \( \mathcal{A} \) of the well-known Ackermann function defined by

\[
\begin{align*}
\mathcal{A}(0)(y) &= 1 + y \\
\mathcal{A}(x + 1)(y) &= \mathcal{A}(x)(\mathcal{A}(y) + 1)(y)
\end{align*}
\]

where for unary \( f, f^{(k)} \) denotes the \( k \)th iterate of \( f \), that is, \( f^{(0)}(x) = x \) and \( f^{(k+1)}(x) = f(f^{(k)}(x)) \). To see this, first observe that the iteration functional \( I \) satisfying

\[
I(x, f, y) = f^{(x)}(y)
\]

can be defined as \( I = R_{(\iota \to \iota) \to \iota \to \iota}(G, H) \) with \( G, H \) defined by \( G(f, y) = y \) and \( H(x, J, f, y) = f(J(f, y)) \).

Now let the functional \( \tilde{H} \) be defined by \( \tilde{H}(x, f, y) = I(y + 1, f, y) \). Then we obtain \( \mathcal{A} = R_{\iota \to \iota}(S, \tilde{H}) \).

In the examples above higher type recursion is only used to define number-theoretical functions, and non-primitive recursive growth rate is easily obtained. It is well-known that by recursion in all finite types one can define exactly the so-called \( \epsilon_0 \)-recursive functions, that is, functions defined by recursion along a well-ordering of the natural numbers of an order type less to the ordinal \( \epsilon_0 \) (cf. Schwichtenberg [57]).

From the examples above we see that while recursion equations for functions are easy to grasp, there is some work involved to represent functions in the form \( R_{\sigma}(G, H) \). So one would wish a calculus which facilitates the representation of functionals in a more direct way, using expression-like explicit notation as common in functional programming languages, and moreover, a calculus which allows one to perform computations. This is the point where system \( T \) proves fruitful.

Objects in system \( T \) are called terms, each of them having a unique type which can be read off the term. Terms of type \( \sigma \), denoted \( s^{\sigma}, r^{\sigma}, t^{\sigma} \), are inductively built from variables \( x^{\sigma} \) and constant symbols

\[
\begin{align*}
0 & \quad \text{of type } \iota \\
S & \quad \text{of type } \iota \to \iota \\
R_{\sigma} & \quad \text{of type } (\iota \to \sigma \to \sigma) \to \iota \to \sigma 
\end{align*}
\]

(for zero)

(for successor \( S \))

(for recursion in type \( \sigma \))
by lambda abstraction $(\lambda x^\sigma.r^\rho)^{\sigma\rightarrow\rho}$ and application $(r^{\sigma\rightarrow\rho}s^\sigma)^\rho$.

Type information is omitted when clear from the context, or of no interest. Following the conventions for types, terms of the form $(\ldots((rs_1)s_2)\ldots)s_n$ are usually written as $rs_1s_2\ldots s_n$, or just as $rs$. As well, terms of the form $\lambda x_1.\ldots\lambda x_{n+1}.r$ are abbreviated by $\lambda x_1\ldots x_{n+1}.r$.

An occurrence of a variable $x^\sigma$ in a term $r$ is called free if it is not in $r'$ of a “subterm” $\lambda x^\sigma.r'$ of $r$; all other occurrences of $x^\sigma$ in $r$ are called bound. Terms without free occurrences of variables are called closed.

Intuitively, closed terms $t^\sigma$ denote functionals of type $\sigma$. To make the construction work, however, one must define for arbitrary terms $t$ the functional $t$ denotes in an environment $\varphi$, denoted by $[t]_\varphi$, thus giving the denotational semantics for system $T$. An environment is a type respecting mapping from the set of variables into $\bigcup_\varphi H_\sigma$. Given $\varphi$, $[t]_\varphi$ is inductively defined by:

$$[x^\sigma]_\varphi := \varphi(x^\sigma)$$
$$[0]_\varphi := 0$$
$$[S]_\varphi := S$$

$$[R_\sigma]_\varphi(g,h) := R_\varphi(g,h) \quad \text{for } g \in H_\sigma, h \in H_{\rightarrow\sigma\rightarrow\sigma}$$

$$[r^{\sigma\rightarrow\rho}s^\rho]_\varphi := [r]_\varphi([s]_\varphi)$$

$$[\lambda x^\sigma.r]_\varphi(f) := [r]_\varphi[f\leftarrow x]$$

where $\varphi[x \leftarrow f]$ denotes the environment which results from $\varphi$ by altering $\varphi$ at $x$ to $f$. For closed terms $t$, the functional $[t]_\varphi$ is independent of the environment $\varphi$, so one just writes $[t]_\varphi$.

One can easily verify that the example functions above do have the following representations in $T$:

$$t_{\text{add}} := \lambda x^1.y^1.R_\sigma y(\lambda u^1.v^1.Sv)x$$

$$t_{\text{mult}} := \lambda x^1.y^1.R_\sigma 0(\lambda u^1.v^1.t_{\text{add}} y v)x$$

$$t_e := R_{\lambda \rightarrow 1}S(\lambda u^1.V^1\leftarrow i_1 y^1.V(V y))$$

$$t_A := R_{\lambda \rightarrow 1}S(\lambda u^1.V^1\leftarrow i_1 y^1.R_1 y(\lambda u^1.v^1.V v)(S y))$$

Computation in system $T$ is given by an operational semantics, roughly speaking a method of transforming a given term $t$ into a representation $\text{nf}(t)$ without “detours” called normal form of $t$, that is, a term to which none of the rules in this transformation process is applicable.

Intuitively, by lambda abstraction $\lambda x^\sigma.r$, where $x^\sigma$ may or may not have free occurrences in $r^\rho$, we define an input position, $\preceq$, of the functional that $r$ represents. Given a “functional” $s^\sigma$, we can input this functional to $\lambda x^\sigma.r$ by forming the application term $(\lambda x^\sigma.r)s$, the idea being that in the computation process, each free occurrence of $x^\sigma$ in $r$ is (simultaneously) replaced with $s$. Of course, in that replacement, one must make sure that no free occurrence of a variable in $s$ becomes bound after the replacement – otherwise the denotation might change. As the denotation of a term does not depend on the “names” of the bound variables, this can always be ensured by bound renaming: let $r[s/x^\sigma]$ denote the result of that replacement.

This replacement is called $\beta$-conversion, and is denoted by:

$$(\lambda x^\sigma.r)s \mapsto r[s/x^\sigma]$$

It is the only computation rule for the simply typed lambda calculus. Reading the equation rules for recursion
in type $\sigma$ as an operation on terms, one obtains the following $R_\sigma$-conversion rules:

$$R_\sigma g h \, 0 \mapsto g$$
$$R_\sigma g h \,(S\, n) \mapsto h \, n \,(R_\sigma g h n)$$

Observe that due to the binding mechanism (lambda abstraction), the type of a constant $R_\sigma$ is minimal.

Now, given a program in $T$, that is, a closed term $t$ of type $\vec{s} \rightarrow \iota$, and given “input data” being numerals $\vec{n}$, by performing $\beta$ and $R$-conversions in whatever order to the initial (closed ground type) term $t\vec{n}$, after finitely many steps, one obtains the normal form $nf(t\vec{n})$, a numeral denoting $[t\vec{n}]$.

This follows from a general result usually referred to as “strong normalisation” stated below. We write $t \rightarrow t'$ if $t'$ results from $t$ by converting one subterm of $t$ according to $\rightarrow$. We say $t$ reduces to $t'$ if $t \rightarrow^* t'$ where $\rightarrow^*$ denotes as usual the reflexive and transitive closure of $\rightarrow$. Observe that $\rightarrow$ is correct, that is, if $t \rightarrow^* t'$ then $t,t'$ denote the same functional in every environment. Furthermore, $\rightarrow$ is consistent with substitution, that is, if $t \rightarrow^* t'$ and $\vec{s} \rightarrow^* \vec{s}'$ (component-wise), then $t[\vec{s}/\vec{x}] \rightarrow^* t'[\vec{s}'/\vec{x}]$. As usual $t[\vec{s}/\vec{x}]$ denotes the result of simultaneously substituting terms $\vec{s}$ for the free occurrences of variables $\vec{x}$ (of equal types) in $t$, respectively. When writing $t[\vec{s}/\vec{x}]$, through bound renaming, one can always assume that no free occurrence of a variable in $\vec{s}$ becomes bound after the substitution.

By applying a powerful method based on “strong computability predicates” introduced by Troelstra [61] and generalising a method from Tait [60], one obtains the following strong normalisation result: Every maximal reduction sequence $t = t_1 \rightarrow t_2 \rightarrow \ldots$ for a term $t$ is finite and results in a term in normal form. Furthermore, employing a technique from Newman [42], strong normalisation implies that this normal form is unique up to bound renaming. Therefore one writes $nf(t)$ for the normal form of $t$.

So we see that $\iota$ is a data type and normalisation allows one to produce data from programs on input data $\vec{n}$, although higher type objects are allowed in between, both as arguments and results of other higher type objects. Thus, system $T$ lays the foundation of a functional programming language, in which computation is normalisation. As well, every term can be considered an algorithm.

2.2. The Grzegorczyk hierarchy

Grzegorczyk [23] was the first to classify the primitive recursive functions ($P\, R$) by a hierarchy of classes $E_n$. Following a more modern approach from Ritchie [52], the functions at Grzegorczyk level $n$ are built from the initial functions $0, S$, all projections $\Pi^n_i(\vec{x}) = x_i$ and the $n$th Ackermann branch $A_n$ (defined below) by composition and bounded primitive recursion:

- $f(\vec{x})$ is defined by composition from functions $g(\vec{y}), h_1(\vec{x}), \ldots, h_l(\vec{x})$ with $l = \#(\vec{y})$ if for all $\vec{x}$,
  $$f(\vec{x}) = g(\vec{h}_1(\vec{x}), \ldots, \vec{h}_l(\vec{x})).$$

- $f(x, \vec{y})$ is defined by bounded primitive recursion\(^1\) from $g(\vec{y}), h(u, \vec{y}, v), b(x, \vec{y})$ if for all $x, \vec{y}$,
  $$f(0, \vec{y}) = g(\vec{y})$$
  $$f(x + 1, \vec{y}) = h(x, \vec{y}, f(x, \vec{y}))$$
  $$f(x, \vec{y}) \leq b(x, \vec{y}).$$

\(^1\)Note that in the presence of projections and closure under composition, a function can be defined by recursion on any input position.
The Grzegorczyk hierarchy

Thus, the idea in the design of each class $E_n$ is that primitive recursion can be used to define new functions at level $n$ as long as they are bounded by functions already defined at level $n$. That this construction actually leads to new functions at every level is due to the presence of the functions $A_n$ at level $n$. The Ackermann branches $A_n(x, y) = A(n, x, y)$ refer to the Ackermann function $A$ defined by:

$$A(0, x, y) := y + 1$$

$$A(n + 1, x, 0) :=
\begin{cases}
  x & n = 0 \\
  0 & n = 1 \\
  1 & \text{else}
\end{cases}$$

$$A(n + 1, x, y + 1) := A(n, x, A(n + 1, x, y))$$

Note that $A_1(x, y) = x + y$, $A_2(x, y) = x \cdot y$, $A_3(x, y) = x^y$, $A_4(x, y) = x^{x^{\cdots}}$ with $y$ occurrences of $x$, and so forth. The functions $A_n$ act as the principal functions in terms of growth rate at level $n$: Schwichtenberg [54] proved that for each $f \in E_n$, one can find a constant $c_f$ satisfying $f(\vec{x}) \leq A_{n+1}(\max(2, \vec{x}), c_f)$. In other words, each function at level $n$ can be bounded by a fixed number of iterations of $A_n$.

Accordingly, functions in $E_0$ have “constant growth”, functions in $E_1$ have “linear growth”, $E_2$ characterises “polynomial growth”, and according to Ritchie [51], this class is identical to the class FLINSPACE of functions computable in binary on a Turing machine in linear space, $E_3$ classifies (iterated) “exponential growth” known to be equivalent to the Kalmár-elementary functions ([53]), and so forth.

Built in primitive recursion is the mechanism for decrementing the recursion argument or for case analysis of whether the recursion argument is zero, e.g. predecessor $P(x)$ and conditional $C(x, y, z)$ are defined by:

$$P(0) = 0 \quad C(0, y, z) = y$$

$$P(x + 1) = x \quad C(x + 1, y, z) = z$$

These are examples of flat recursions, that is, recursions which do not use any of their computed values. Observe that the predecessor is in $E_0$, but the conditional is a proper element of $E_1$. The latter implies that each class $E_{n+1}$ is closed under “definition by cases”.

The Grzegorczyk classes (at and above linear-space level) are closed under various modes of construction other than bounded primitive recursion, some of which are listed below.

Functions $f_1, \ldots, f_k$ are defined by simultaneous recursion from $g_1, \ldots, g_k, h_1, \ldots, h_k$ if for all $\vec{x}, y$,

$$f_i(\vec{x}, 0) = g_i(\vec{x})$$

$$f_i(\vec{x}, y + 1) = h_i(\vec{x}, y, f_1(\vec{x}, y), \ldots, f_k(\vec{x}, y)).$$

If in addition each $f_i$ is bounded by a function $b_i$, that is, $f_i(\vec{x}, y) \leq b_i(\vec{x}, y)$ for all $\vec{x}, y$, then $f$ is said to be defined by bounded simultaneous recursion from $g_1, \ldots, g_k, h_1, \ldots, h_k, b_1, \ldots, b_k$.

Now each class $E_{n+2}$ is closed under bounded simultaneous recursion. But observe that one application of unbounded simultaneous recursion from functions in $E_{n+2}$ yields functions in $E_{n+3}$.

Let $\text{Rel}(E_n)$ denote the set of relations $R \subseteq \mathbb{N}^k$ with characteristic function $c_R : \mathbb{N}^k \to \mathbb{N}$ in $E_n$, where

$$c_R(\vec{x}) :=
\begin{cases}
  1 & \text{if } R(\vec{x}) \\
  0 & \text{else}
\end{cases}$$

While closure under bounded simultaneous recursion is only obtained at and above linear-space level, each class $\text{Rel}(E_n)$ is closed under propositional logic and bounded quantification, that is,
if $R, Q$ are in Rel($\mathcal{E}_n$), then so are $¬R, R \land Q, R \lor Q$, 

if $R \subseteq \mathbb{N}^{k+1}$ is in Rel($\mathcal{E}_n$), then so are $\forall i \leq x. R(i, \vec{y})$ and $\exists i \leq x. R(i, \vec{y})$.

Furthermore, each class $\mathcal{E}_n$ is closed under bounded minimisation, that is, if $g$ is in $\mathcal{E}_n$, then so is $f$ satisfying

$$f(x, \vec{y}) = (\mu i \leq x)[R(i, \vec{y})] := \begin{cases} \min\{i \leq x \mid g(i, \vec{y}) = 0\} & \text{if such exists} \\ x & \text{else.} \end{cases}$$

Observe that one could equivalently use the alternative value 0 instead of $x$.

As much as Grzegorczyk’s approach is satisfactory from a mathematical point of view and proved fruitful in logic (cf. e.g. [53], [57], [12], [18], [24]) and in characterising other complexity classes (cf. e.g. [15]) by function algebras (such as Cobham’s [16] characterisation of the class FPTIME of the polynomial-time computable functions by a function algebra $FP$), it is of no help in understanding the impact of nesting recursions on the running time.

### 2.3. A version of the Grzegorczyk hierarchy

Another version of the Grzegorczyk hierarchy (cf. Rose [53] or Clote [15]) at and above linear-space level is obtained when the principal functions are based on “pure iteration”.

The principal functions $E_1, E_2, E_3, \ldots$ are defined as follows:

$$E_1(x) := x^2 + 2$$

$$E_{n+2}(x) := E_{(x)}^{(2)}(2)$$

These functions $E_n$ feature the following (easily provable) monotonicity properties. For all $n, x, t$, one has:

$$x + 1 \leq E_{n+1}(x)$$

$$E_{n+1}(x) \leq E_{n+1}(x + 1)$$

$$E_{n+1}(x) \leq E_{n+2}(x)$$

$$E_{n+1}^{(t)}(x) \leq E_{n+2}(x + t)$$

The $n$th Grzegorczyk class $\mathcal{E}^n$, $n \geq 2$, is the least class of functions containing the initial functions zero, successor, projections, maximum and $E_{n-1}$, and is closed under composition and bounded recursion.

Observe that the additional initial function max is necessary to allow multiple variable functions to occur freely in the hierarchy. Furthermore, for those principal functions one easily verifies that each $f \in \mathcal{E}^{n+1}$ satisfies $f(\vec{x}) \leq E_{n}^{(c_f)}(\vec{x})$ for some constant $c_f$. Finally, it is not too difficult to show $\mathcal{E}^n = \mathcal{E}_n$ for $n \geq 2$.

### 2.4. Recursion on notation and a binary version of system $T$

Another form of system $T$ is obtained when replacing the natural numbers in unary representation by the natural numbers in binary representation, $\text{bin}(x)$ for $x \in \mathbb{N}$ (with $\text{bin}(0) = \varepsilon$, the empty string).

Similar to Ritchie’s [51] characterisation of FLINSpace by $\mathcal{E}_2$, the background is that one would like to achieve a machine-independent, purely functional characterisation of the class FPTIME of number-theoretical functions computable on a Turing machine in time polynomial in the binary length $|x|$ of the
inputs. One first simulates binary representations \( \text{bin}(x) \) by the initial functions 0 and the binary successors \( s_0, s_1 \) satisfying:

\[
\begin{align*}
\text{s}_0(x) &= 2 \cdot x \quad \text{(operation } \text{bin}(x) \mapsto \text{bin}(x)0 \text{ for } x \neq 0) \\
\text{s}_1(x) &= 2 \cdot x + 1 \quad \text{(operation } \text{bin}(x) \mapsto \text{bin}(x)1) \\
\end{align*}
\]

This “data structure” gives rise to a canonical recursion scheme: \( f \) is defined by recursion on notation from functions \( g(\vec{y}), h_0(u, \vec{y}, v), h_1(u, \vec{y}, v) \) if for all \( x, \vec{y}, y, s \):

\[
\begin{align*}
\text{f}(0, \vec{y}) &= g(\vec{y}) \\
\text{f}(s_1(x), \vec{y}) &= h_i(x, \vec{y}, f(x, \vec{y})) \quad \text{for } s_i(x) \neq 0.
\end{align*}
\]

Observe that \( \text{bin}(y) = b_1 \ldots b_l \) with \( b_l = 1 \) implies \( y = s_{b_0}(s_{b_2}(\ldots s_{b_l}(0)\ldots)) \). Thus, for recursion on notation, the recourse is from \( b_1 \ldots b_l \) to \( b_1 \ldots b_2 \) to \( b_1 \ldots b_3 \) to \ldots to \( b_l = s_1(0) \), and finally from \( s_1(0) \) to 0. Thus, one needs \( |x| \) recursive calls of \( f \) when computing \( f(x, \vec{y}) \), where as usual \( |x| = \lceil \log_2(x + 1) \rceil \).

Accordingly, \text{string concatenation} \( \oplus \) satisfying \( x \oplus y = x \cdot 2^{|y|} + y \), or in other words, \( \text{bin}(x \oplus y) = \text{bin}(x)\text{bin}(y) \), can be defined by the obvious recursion on notation (in the second argument):

\[
\begin{align*}
x \oplus 0 &:= x \\
x \oplus s_i(y) &:= s_i(x \oplus y)
\end{align*}
\]

Furthermore, building on \( \oplus \), \text{string multiplication} \( \otimes \) satisfying \( x \otimes y = x \cdot \sum_{i < |y|} 2^{|x|} \), or in other words, \( \text{bin}(x \otimes y) = \text{bin}(x) \ldots \text{bin}(x) (|y| \text{ times}) \), can be defined by:

\[
\begin{align*}
x \otimes 0 &:= 0 \\
x \otimes s_i(y) &:= (x \otimes y) \oplus x
\end{align*}
\]

Similar to primitive recursion, built in recursion on notation is the mechanism for decrementing the recursion argument and for case analysis whether the recursion argument is even. Thus, the binary predecessor \( p(x) = \lfloor \frac{x}{2} \rfloor \), and the binary conditional \( c \) satisfying \( c(s_i(x), y_0, y_1) = y_i \) can be defined by:

\[
\begin{align*}
p(0) &:= 0 \quad c(0, y_0, y_1) := y_0 \\
p(s_i(x)) &:= x \quad c(s_i(x), y_0, y_1) := y_i \quad (\text{for } s_i(x) \neq 0)
\end{align*}
\]

\textbf{Recursion on notation is as strong as primitive recursion.} To see this, first consider the following \text{string exponentiation} \( \circ \) satisfying \( \circ(x) = 2^x - 1 \). It can be defined by recursion on notation from string concatenation:

\[
\begin{align*}
\circ(0) &:= 0 \\
\circ(s_0(x)) &:= \circ(x) \oplus \circ(x) \\
\circ(s_1(x)) &:= s_1(\circ(x) \oplus \circ(x))
\end{align*}
\]

Note that \( |\circ(x)| = x \), and hence \( p(\circ(x)) = \circ(x - 1) \) (where \( \dot{-} \) as usual denotes \text{modified subtraction} with \( x \dot{-} y = 0 \) for \( x \leq y \)). Thus \( \circ \) can be used to synchronise the number of steps in a recursion on notation on \( \circ(x) \) with the number of steps in a primitive recursion on \( x \). This allows primitive recursion to be reduced
to recursion on notation as follows: For every primitive recursive function \( f(\vec{x}) \), one can find a function \( f^*(\vec{x}) \) defined from the initial functions 0, \( s_0, s_1, \Pi^m_i \) by composition and recursion on notation, such that

\[
(\ast) \quad \circ(f(\vec{x})) = f^*(\circ(\vec{x})) \quad \text{for all } \vec{x}.
\]

Thus, once \( x \mapsto |x| \) is defined by recursion on notation, we obtain \( f(\vec{x}) = |f^*(\circ(\vec{x}))| \) as required. The proof of \((\ast)\) is by induction on the structure of \( f(\vec{x}) \). If \( f \) is \( 0, S, \Pi^m_i \) respectively, for \( \circ(0) = 0 \) and \( \circ(x+1) = 2(2^x-1)+1 = s_1(\circ(x)) \). Otherwise the induction hypothesis is used to proceed by:

- If \( f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_l(\vec{x})) \), then let \( f^* \) be defined by \( f^*(\vec{x}) := g^*(h_1^*(\vec{x}), \ldots, h_l^*(\vec{x})) \).

- If \( f \) is defined by primitive recursion from \( g, h \), define \( f^* \) by recursion on notation from \( g^*, h^*, h^* \).

Now \((\ast)\) is obvious in the case of composition. In case of primitive recursion, one proceeds by side induction on \( x \) showing \( f^*(\circ(x, \vec{y})) = \circ(f(x, \vec{y})) \). The base case follows from \( \circ(0) = 0 \) and the induction hypothesis on \( g \), for \( f^*(\circ(0, \vec{y})) = g^*(\circ(\vec{y})) = \circ(g(\vec{y})) = \circ(f(0, \vec{y})) \). The step case is concluded as follows:

\[
\circ(f(x+1, \vec{y})) = \circ(h(x, \vec{y}), f(x, \vec{y})) = h^*(\circ(x, \vec{y}), f(x, \vec{y})) = f^*(\circ(x+1, \vec{y})) \quad \text{by definition}
\]

\[
\circ(f(x+1, \vec{y})) = h^*(\circ(x, \vec{y}), f(x, \vec{y})) = f^*(\circ(x+1, \vec{y})) \quad \text{by the I.H. on } h
\]

\[
\circ(f(x+1, \vec{y})) = h^*(\circ(x, \vec{y}), f^*(\circ(x, \vec{y}))) = f^*(\circ(x+1, \vec{y})) \quad \text{by the side I.H. on } x
\]

The missing definition of \(|\cdot|\) by recursion on notation can be read from the following recursion equations:

\[
\begin{align*}
geven(0) &= 1 \\
even(x+1) &= C(even(x), 1, 0) \\
p(0) &= 0 \\
p(x+1) &= C(even(x+1), p(x), p(x)+1).
\end{align*}
\]

Hence \( (x, y) \mapsto p^{(x)}(y) \) and \(|\cdot|\) are primitive recursive, the latter by \( |x| = (\mu n \leq x)[p^{(n)}(x) = 0] \) and closure under bounded minimisation. Finally, as modified subtraction \( x - y = P^{(y)}(x) \) is primitive recursive, then so are the functions \( c_{\leq}, \text{odd}(x) = 1 - \text{even}(x), \) and \( e(x, y) := p^{(|x|-y)}(x) \). In fact, all of these functions are in Grzegorczyk class \( \mathcal{E}_1 \).

Now suppose that \( f(\vec{x}, y) \) is defined by recursion on notation from functions \( g, h_0, h_1 \), say \( f(\vec{x}, 0) = g(\vec{x}) \) and \( f(\vec{x}, s_i(y)) = h_i(\vec{x}, y, f(\vec{x}, y)) \) for \( s_i(y) \neq 0 \). Assume inductively that \( g, h_0, h_1 \) are primitive recursive. Then the reduction works as follows: First we simulate \( f \) by primitive recursion up to \(|y|\), that is, we define by primitive recursion a function \( \hat{f} \) satisfying \( \hat{f}(\vec{x}, y, z) = f(\vec{x}, e(y, z)) \) for \( z \leq |y| \):

\[
\begin{align*}
\hat{f}(\vec{x}, y, 0) &= g(\vec{x}) \\
\hat{f}(\vec{x}, y, z+1) &= \begin{cases} 
  h_0(\vec{x}, e(y, z), \hat{f}(\vec{x}, y, z)) & \text{if } z+1 \leq |y| \land \text{even}(e(y, z+1)) \\
  h_1(\vec{x}, e(y, z), \hat{f}(\vec{x}, y, z)) & \text{if } z+1 \leq |y| \land \text{odd}(e(y, z+1)) \\
  \hat{f}(\vec{x}, y, z) & \text{else}
\end{cases}
\end{align*}
\]
By induction on \( z \), one easily obtains \( \hat{f}(x, y, z) = f(x, e(y, z)) \) for \( z \leq |y| \), using \( p(e(y, z + 1)) = e(y, z) \) in the step case. This implies \( \hat{f}(x, y, z) = f(x, y) \) for \( z \geq |y| \), and hence \( f(x, y) = \hat{f}(x, |y|) \), concluding the proof that recursion on notation is as strong as primitive recursion.

In the presence of \( p, c \) as initial functions, it can be easily verified that a function \( f \) is defined by recursion on notation from \( g, h_0, h_1 \) if and only if \( f \) can be defined by \textit{modified recursion on notation} from \( g, h \), i.e.,

\[
\begin{align*}
    f(0, \vec{y}) &= g(\vec{y}) \\
    f(s_i(x), \vec{y}) &= h(s_i(x), \vec{y}, f(p(x), \vec{y})) & \text{for } s_i(x) \neq 0.
\end{align*}
\]

Based on modified recursion on notation, one obtains a “binary version” of system \( T \) by adding to the simply typed lambda calculus the following \textbf{constant symbols}:

\[
\begin{align*}
    0 & \quad \text{of type } \iota & (\text{for zero}) \\
    s_0 & \quad \text{of type } \iota \rightarrow \iota & (\text{for binary successor } s_0) \\
    s_1 & \quad \text{of type } \iota \rightarrow \iota & (\text{for binary successor } s_1) \\
    p & \quad \text{of type } \iota \rightarrow \iota & (\text{for binary predecessor } p) \\
    c_\sigma & \quad \text{of type } \iota \rightarrow \sigma \rightarrow \sigma \rightarrow \sigma & (\text{for binary cases in type } \sigma) \\
    \RN_\sigma & \quad \text{of type } \sigma \rightarrow (\iota \rightarrow \sigma \rightarrow \sigma) \rightarrow \iota \rightarrow \sigma & (\text{for recursion on notation in type } \sigma)
\end{align*}
\]

The denotational semantics for this version of system \( T \) is then as expected. In terms of the \textit{operational semantics}, one has to make sure that ground data is uniquely represented, and that the conversion rules for the constant symbols only apply to such unique representations, called binary numerals.

A \textbf{binary numeral} is either \( 0 \) or of the form \( s_{i_1} \ldots s_{i_k} 0 \), where \( i_j \in \{0, 1\} \) for \( j = 1, \ldots, k, k \geq 0 \). In the conversion rules below we assume that \( s_i n \) is a binary numeral (distinct from \( 0 \)). The \textit{conversion rules} for the constant symbols above are then as expected:

\[
\begin{align*}
    s_0 0 & \rightarrow 0 & c_\sigma 0 r t_0 t_1 & \rightarrow r \\
    p 0 & \rightarrow 0 & c_\sigma (s_i n) r t_0 t_1 & \rightarrow t_i \\
    p(s_i n) & \rightarrow n & \RN_\sigma g h 0 & \rightarrow g \\
                  & & \RN_\sigma g h (s_i n) & \rightarrow h (s_i n) (\RN g h n)
\end{align*}
\]

\section{2.5. Cobham’s class \( \text{FP} \)}

As just seen, recursion on notation is as strong as primitive recursion. Hence, in order to characterise the class \( \text{FPTIME} \) by recursion on notation, we must restrict the use of recursion.

The first such restriction is from Cobham [16] who showed \( \text{FPTIME} = \text{FP} \) where functions in \( \text{FP} \) are inductively defined from the \textit{initial functions} \( 0, s_0, s_1 \), all projections \( \pi_i^n \) and \textit{smash} \( x, y = 2^{|x|+|y|} \) by composition and bounded recursion on notation, that is, \( f(x, \vec{y}) \) is defined by \textit{bounded recursion on notation} from \( g(\vec{y}), h_0(u, \vec{y}, v), h_1(u, \vec{y}, v), b(x, \vec{y}) \) if for all \( x, \vec{y} \):

\[
\begin{align*}
    f(0, \vec{y}) &= g(\vec{y}) \\
    f(s_i(x), \vec{y}) &= h_i(x, \vec{y}, f(x, \vec{y})) & \text{for } s_i(x) \neq 0 \\
    f(x, \vec{y}) &\leq b(x, \vec{y})
\end{align*}
\]

Similar to the design of the Grzegorczyk classes, the idea is that recursion on notation can be used as long as the resulting function is bounded by a function already defined in the class. To make this idea work, the “principal function” \textit{smash} is added to the initial functions.
Observe that by a straightforward induction on the structure of functions in FP, one can show that each \( f(\vec{x}) \) has a polynomial length-bound, referred to as FP-\textbf{Bounding}, that is, a polynomial \( p_f(\vec{x}) \) satisfying
\[
|f(\vec{x})| \leq p_f(|\vec{x}|).
\]
Now we see that the bounding condition in bounded recursion on notation can be equivalently stated as
\[
|f(x, \vec{y})| \leq |b(x, \vec{y})|
\]
for one also obtains a polynomial length-bound \( p_f \) for each \( f \) in the modified class, implying \( f(\vec{x}) \leq 2^{p_f(|\vec{x}|)} \), and for polynomials \( q(\vec{x}) \) the function \( f_q(\vec{x}) = 2^{q(|\vec{x}|)} \) is in FP, referred to as FP-\textbf{Closure}.

To see this, we proceed by induction on the structure of polynomials \( q(\vec{x}) \). The base cases where \( q(\vec{x}) \) is \( x_i \) or a constant are obvious. For the step case, suppose that \( q(\vec{x}) = q_1(\vec{x}) \circ q_2(\vec{x}) \) with \( \circ \in \{+,-,\cdot\} \). First recall that concatenation \( \oplus \) satisfies \(|x \oplus y| = |x| + |y|\). Hence \( x \oplus y \leq 2^{|x| + |y|} \leq \text{smash}(s_1(x), s_1(y)), \) implying that \( \oplus \) can be defined in FP by bounded recursion on notation. Thus, we obtain \( f_q \in \text{FP} \) from the induction hypothesis on \( q_1, q_2 \), because we have \(|p(2^x)| = x\), implying
\[
\begin{align*}
2^{q_1(|\vec{x}|) + q_2(|\vec{x}|)} &= \text{smash}(1, p(2^{q_1(|\vec{x}|)}) \oplus p(2^{q_2(|\vec{x}|)})) \\
2^{q_1(|\vec{x}|) \cdot q_2(|\vec{x}|)} &= \text{smash}(p(2^{q_1(|\vec{x}|)}), p(2^{q_2(|\vec{x}|)})).
\end{align*}
\]

### 2.6. Heinermann’s classes based on nesting depth of recursions

Heinermann [25] was the first to classify the \( PR \)-functions without explicitly restricting the use of recursion. His approach consists in counting the nesting depth of recursions needed to define a function \( f \) at level \( n \) denoted \( \mathcal{R}_n \): \( f \) belongs to level \( n \) if it is an initial function \( 0, S, \Pi^m_j \) or obtained by composition from functions at level \( n \), or else \( n > 0 \) and \( f \) is obtained by one primitive recursion from functions at level \( n - 1 \). Here the Ackermann branches are not present as initial functions; each \( A_n \) can be defined in \( \mathcal{R}_n \). So the \( n \)th Heinermann class \( \mathcal{R}_n \) is based on definitions of primitive recursive functions, and nesting depth is a measure of the amount of work that, implicitly, must be performed in order to produce a value.

In the context of the subsystem \( PR_1 \) of Gödel’s \( T \) consisting of the terms using only ground type variables \((x,y,z,\ldots)\) and recursion in type \( t \), the nesting depth \( \text{deg}(t) \) of a term \( t \in PR_1 \) is inductively defined by:

\[
\begin{align*}
\text{deg}(u) := & 0 \quad \text{for } u \text{ among } x, 0, S \\
\text{deg}(\lambda x.r) := & \text{deg}(r) \\
\text{deg}(rs) := & \max\{\text{deg}(r), \text{deg}(s)\} \\
\text{deg}(R, g h) := & \max\{\text{deg}(g), 1 + \text{deg}(h)\}
\end{align*}
\]

Under these definitions, we can restate the \( n \)th Heinermann class as
\[
\mathcal{R}_n = \{ f \in \mathcal{PR} \mid f = [t] \text{ for some closed } t \in PR_1 \text{ with } \text{deg}(t) \leq n\}.
\]

The measure \( \text{deg} \) is related to the Grzegorczyk hierarchy at and above the Kalmár-elementary level: Schwichtenberg [54] established \( \mathcal{E}_{n+1} = \mathcal{R}_n \) for \( n \geq 3 \), and Müller [41] improved it by showing \( \mathcal{E}_3 = \mathcal{R}_2 \).

However, the measure \( \text{deg} \) fails to classify sub-elementary complexity classes such as \( \text{FLINSPACE} \) or \( \text{FPTIME} \). Typical of this failure is that \( \text{deg} \) cannot separate the definition of multiplication from the natural definition of exponentiation. To see this, recall the definition of addition:

\[
\begin{align*}
\text{add}(0, y) &= y \\
\text{add}(x + 1, y) &= \text{add}(x, y) + 1
\end{align*}
\]
Now consider the following definitions of multiplication and exponentiation:

\[
\begin{align*}
\text{mult}(0, y) &= 0 \\
\text{mult}(x + 1, y) &= \text{add}(y, \text{mult}(x, y))
\end{align*}
\]

\[
\begin{align*}
\exp(0) &= 1 \\
\exp(x + 1) &= \text{add}(\exp(x), \exp(x))
\end{align*}
\]

Both functions are defined by one recursion from addition, hence both definitions have nesting depth 2. How does this explosion in computational complexity come about? The gist of the matter lies in different “control” of the input positions: In either case the “input position” \( x \) controls a recursion, but while mult passes its computed values to the “input position” \( y \) of add having no control over any other recursion, each computed value of \( \exp \) “controls” the recursion add is defined by.

2.7. Simmons’ approach to restrict recursion in type \( \iota \rightarrow \iota \)

Simmons [59] was the first to give syntactic restrictions to the use of recursion in type \( \iota \rightarrow \iota \) so as to classify the primitive recursive functions. Primarily interested in finding an illuminating proof of the well-known fact (extensively elaborated in Rózsa Péter’s classic book [50]) that the primitive recursive functions are closed under many more complex modes of construction other than primitive recursion, Simmons defined a type level two function algebra with only ground type recursion, but which is closed under recursion in \( \iota \rightarrow \iota \).

Simmons’ key observation was that the use of recursion in type \( \iota \rightarrow \iota \) can only lead to non-primitive growth rate if “dormant” input positions in a function definition are mixed with “active” ones, that is, input positions which control a recursion. To exemplify this, recall the definition of function \( \varTheta \):

\[
\begin{align*}
\varTheta(0)(y) &= 1 + y \\
\varTheta(x + 1)(y) &= \varTheta(x)(y + 1)(y) = I(y + 1, \varTheta(x), y)
\end{align*}
\]

Here the first “input position” \( x \) is active, for it controls the outermost recursion. But the second “input position” \( y \), a “parameter” with respect to that definition, is both active and dormant, for in the recursion step it controls the recursion responsible for the iteration of computed functionals \( \varTheta(x) \), and it is a pure parameter in the definition of this iteration.

This gave rise to a type level two functional algebra \( S(\varTheta) \), in which functionals come in the form \( F(\vec{x}; \vec{y}; \vec{f}) \), with \( \vec{x} \) thought of as ground type active input positions, and \( \vec{y}, \vec{f} \) as ground type, \( \iota \rightarrow \iota \) dormant input positions respectively, and where \( \varTheta(\vec{x}; \vec{y}; \vec{f}) \) is an arbitrary but fixed functional\(^2\). Using this bookkeeping, functionals in \( S(\varTheta) \) are defined inductively by the initial functionals \( \varTheta \), all zero functionals \( 0(\vec{x}; \vec{y}; \vec{f}) = 0 \), all successor functionals \( S_j(\vec{x}; \vec{y}; \vec{f}) = S(y_j) \), all projections \( \Pi_j(\vec{x}; \vec{y}; \vec{f}) = y_j \), all application functionals \( \text{App}_{j,k}(\vec{x}; \vec{y}; \vec{f}) = f_k(j) \) by allowable composition and allowable recursion, that is,

- \( F(\vec{x}; \vec{y}; \vec{f}) \) is defined by allowable composition from \( G(\vec{u}; \vec{v}; \vec{g}), \vec{A}(\vec{x}; ;), \vec{B}(\vec{x}; \vec{y}; \vec{f}), \vec{C}(\vec{x}; y; \vec{f}) \) if
  \[
  F(\vec{x}; \vec{y}; \vec{f}) = G(\ldots, A_i(\vec{x}; ;), \ldots, B_j(\vec{x}; \vec{y}; \vec{f}), \ldots, \ldots, \lambda y.C_k(\vec{x}; y; \vec{f}), \ldots)
  \]

- \( F(\vec{x}; \vec{y}; \vec{f}) \) is defined by allowable recursion from \( G(\vec{x}; \vec{y}; \vec{f}), H(u, \vec{x}; \vec{y}; v; \vec{f}) \) if
  \[
  \begin{align*}
  F(0, \vec{x}; \vec{y}; \vec{f}) &= G(\vec{x}; \vec{y}; \vec{f}) \\
  F(x + 1, \vec{x}; \vec{y}; \vec{f}) &= H(x, \vec{x}; \vec{y}; F(x, \vec{x}; \vec{y}; \vec{f}); \vec{f})
  \end{align*}
  \]

\(^2\)The presence of “\( \varTheta \)” can be explained by Simmons’ interest in investigating the limits of the notion “primitive recursive in”.\]
So recursion is restricted in that the computed values $F(x, \bar{x}; \bar{y}; \bar{f})$ can only be passed to dormant input positions ($v$) of the “step functional” $H(u, \bar{x}; \bar{y}; v; \bar{f})$. In terms of allowable composition, while recent research [45, 46, 6] shows that it is irrelevant whether a value passed on to an active input position depends on dormant input positions, one cannot drop the requirement that the lambda abstracted input position in every substituted functional $\lambda y.C_k(x; y; \bar{f})$ is dormant, and that the resulting function may only depend on active input positions (cf. [7]). Simmons essentially proved the following:

1. Every $F$ in $\mathcal{S}(\Theta)$ can be defined from $\Theta$ using recursion in type $\iota \rightarrow \iota$. But if $\Theta$ is a function $\Theta(\bar{x}; \bar{y}; \bar{f})$ with ground type inputs only, then every $F$ in $\mathcal{S}(\Theta)$ is primitive recursive in $\Theta$, that is, it can be defined from $\Theta$ using ground type recursion only.

2. If $A(; y;)$ and $B(x; y; f)$ are in $\mathcal{S}(\Theta)$, then so is $F$ defined from $A, B$ by recursion in type $\iota \rightarrow \iota$:

$$F(0, y) = A(; y;)$$
$$F(x + 1, y) = B(x; y; \lambda y. F(x, y))$$

Now observe that all projections $\Pi_i(\bar{x}; \bar{y}; \bar{f}) = x_i$ can be defined by allowable recursion from zero and successor functionals, for example:

$$\Pi_0(0, \bar{x}; \bar{y}; \bar{f}) = 0$$
$$\Pi_0(x + 1, \bar{x}; \bar{y}; \bar{f}) = S_{\#(\bar{y}) + 1}(x, \bar{x}; \bar{y}, \Pi_0(x, \bar{x}; \bar{y}; \bar{f}); \bar{f})$$

Hence allowable composition can be used to turn every dormant input position into an active one, that is, if $A(\bar{x}; y; \bar{y}; \bar{f})$ is in $\mathcal{F}(\Theta)$, then so is $A^*$ satisfying $A^*(\bar{x}; y; \bar{y}; \bar{f}) = A(\bar{x}; \bar{y}; \bar{f})$. So we see that the iteration functional $I(x, f, y) = f^*(x)(y)$ can be defined in $\mathcal{F}(\Theta)$, but only in the form $I(x; y; f)$. Thus, in $\mathcal{F}(\Theta)$ one can define $I'(y; f) = I(y + 1; y; f)$, but the definition of $A$ is ruled out.

So Simmons’ approach gives insight as to why recursion in type $\iota \rightarrow \iota$ leads to non-primitive recursive growth rate, and how it can be restricted so as to stay in the realm of primitive recursion. However, both the setting of his classes $\mathcal{S}(\Theta)$ and the method he develops to prove the results stated above are unlikely to be generalised to recursion in all finite types. But on closer inspection, one can read from his analysis the following: Typical of many-fold recursions, such as

$$t_A := R_{x \rightarrow v} \ S (\lambda u^v V^{\iota \rightarrow \iota} y^v. R_{x \rightarrow v} (\lambda u^v V y^v) (S y))$$

(and all known versions of the Ackermann function come down to that when written in normal form) is that computed functionals ($V$ above) of an outer recursion are applied to computed functionals ($v$ above) of an inner recursion. In fact, this causes a non-primitive recursive explosion in computational complexity.

### 2.8. The approach of Bellantoni and Cook to restrict recursion on notation

Building on work of Leivant [32, 33], Bellantoni and Cook [5] were the first to give a purely functional characterisation of the polynomial-time computable functions that does away with the “explicit” reference to the growth rate of functions defined by (bounded) recursion on notation in Cobham’s class $\text{FP}$. They observed that this “explicit” reference can be made “implicit” by ensuring that computed values in recursions do not control other recursions. To exemplify this central idea, recall the following definitions:

$$x \otimes 0 = 0$$
$$x \otimes s_i(y) = (x \otimes y) \oplus x$$
$$\odot(0) = 0$$
$$\odot(s_0(x)) = \odot(x) \oplus \odot(x)$$
$$\odot(s_1(x)) = s_1(\odot(x) \oplus \odot(x))$$

(for $x \neq 0$)
Although both functions are defined by one recursion from $\oplus$, string multiplication $\otimes$ is polynomial-time computable, while $\odot$ is a proper Kalmár-elementary function. The reason is simply that each computed value $\odot(x)$ controls the recursion in $\oplus$. In contrast to $\odot$, no computed value in the recursion for $\otimes$ controls another recursion.

The idea that **computed values in recursions must not control other recursions** led to the class $BC$ where each function is written in the form $f(\vec{x}; \vec{y})$, in order to bookkeep and sort out those input positions $\vec{x}$ which may control a recursion from those $\vec{y}$ which do not. Thus, functions in $BC$ are inductively defined from the initial functions $0, s_0(; y), s_1(; y), \pi_i^{n,m}(\vec{x}; \vec{y}), p(; y), c(; x, y, z)$ by safe composition and safe recursion:

- $f(\vec{x}; \vec{y})$ is defined by **safe composition** from $g(\vec{u}; \vec{v}), \vec{g}(\vec{x}; ;), \vec{h}(\vec{x}; \vec{y})$, with $\#(\vec{u}) = \#(\vec{g})$ and $\#(\vec{v}) = \#(\vec{h})$, if for all $\vec{x}, \vec{y}$,
  $$f(\vec{x}; \vec{y}) = g(\vec{g}(\vec{x}; ;); \vec{h}(\vec{x}; \vec{y})).$$

- $f(\vec{x}; \vec{y})$ is defined by **safe recursion** from $g(\vec{x}; \vec{y}), h_0(u, \vec{x}; \vec{y}, v), h_1(u, \vec{x}; \vec{y}, v)$ if for all $x, \vec{x}, \vec{y}$,
  $$f(0, \vec{x}; \vec{y}) = g(\vec{x}; \vec{y})$$
  $$f(s_i(x), \vec{x}; \vec{y}) = h(x, \vec{x}; \vec{y}, f(x, \vec{x}; \vec{y})) \quad \text{for } s_i(x) \neq 0.$$  

Observe the striking similarity to Simmons’ [59] design of his classes $S(\Theta)$.

An easy induction on the structure of $f(\vec{x}; \vec{y})$ shows that each $f$ in $BC$ has a **poly-max length bound**, that is a polynomial $q_f$ satisfying

$$|f(\vec{x}; \vec{y})| \leq q_f(|\vec{x}|) + \max(|\vec{y}|).$$

Using this **Poly-max-length-bounding Lemma**, every recursion in $BC$ can be written as a bounded recursion in Cobham’s class $FP$, implying $BC \subseteq FP$. The converse holds by simulating the functions of $FP$ in $BC$. 

3. The measure $\mu$ on $\lambda$ terms over ground type variables and recursion

In this chapter, two subsystems $\text{PR}_1, \text{PR}_2$ of Gödel’s $T$ with only ground type variables and ground type recursion are considered, where $\text{PR}_1$ is based on primitive recursion and $\text{PR}_2$ on recursion on notation. To each term $t$ in these systems, a measure $\mu(t)$ is assigned in a purely syntactic fashion.

It is shown that the measure $\mu$ on $\text{PR}_1$ is closely related to the Grzegorczyk hierarchy: Programs of $\mu$-measure $n + 1$ exactly compute the functions in Grzegorczyk level $n + 2$. In particular, programs of $\mu$-measure 1 exactly compute the $\text{FLINSPACE}$ functions. The measure $\mu$ on $\text{PR}_2$ characterises the same hierarchy of classes, except that programs of $\mu$-measure 1 precisely compute the $\text{FPTIME}$ functions.

3.1. Intuition

The measure $\mu$ refines nesting depth so as to follow up the “control” of each input position in a computation. Thereby it separates recursions which may cause a blow up in computational complexity, called top recursions, from those which do not, called side recursions.

Intuitively, to each definition of a primitive recursive function $f(x_1, \ldots, x_l)$ one assigns ranks $\tau_1, \ldots, \tau_l$ to all “input positions” $x_1, \ldots, x_l$ where each $\tau_i$ is either a natural number $n_i$ or $\iota$, the idea being that:

- if $\tau_i$ is $n_i$, then input position $x_i$ controls $n_i$ top recursions,
- if $\tau_i$ is $\iota$, then input position $x_i$ is redundant,

and that the maximum of the ranks $n_i$ defines the computational complexity of $f$.

We give the inductive definition of $f(x_1, \ldots, x_l)$ has ranks $\tau_1, \ldots, \tau_l$ in terms of the usual schemes for defining primitive recursive functions. First we need some auxiliary operations. Given a rank $\tau$, we define

$$ R(\tau) := \begin{cases} n & \text{if } \tau = n \\ 0 & \text{else.} \end{cases} $$

Furthermore, we extend max so as to operate on ranks $(\tau, \tau')$, and we define its “strict version” $\text{smax}$:

$$ \text{max}(\iota, \tau) := \begin{cases} n & \text{if } \tau = n \\ \iota & \text{else.} \end{cases} $$

$$ \text{smax}(\tau, \tau') := \begin{cases} \text{max}(\tau, \tau') & \text{if } \tau, \tau' \in \mathbb{N} \\ \iota & \text{else.} \end{cases} $$

The inductive definition of $f(x_1, \ldots, x_l)$ has ranks $\tau_1, \ldots, \tau_l$ is then as follows:

- $f$ is an initial function. No input position of $f$ controls a recursion, but it may or may not be redundant.

  $$ 0 \quad \text{has no rank (it has no input position at all)} $$

  $$ \text{Successor } S(x) \quad \text{has rank 0} $$

  $$ \text{Projection } \pi^j_i(x_1, \ldots, x_l) = x_i \quad \text{has ranks } \iota, \ldots, \iota, 0, \iota, \ldots, \iota \\ \text{(with 0 for the only non-redundant input position } x_i) $$

- $f$ is defined by composition, say $f(\bar{x}) = g(h_1(\bar{x}), h_2(\bar{x}))$ for simplicity. Assume that

  $$ g(y_1, y_2) \quad \text{has ranks } \tau_1, \tau_2 $$

  $$ h_1(x_1, \ldots, x_l) \quad \text{has ranks } \rho_1, \ldots, \rho_l $$

  $$ h_2(x_1, \ldots, x_l) \quad \text{has ranks } \delta_1, \ldots, \delta_l. $$

Then each input position $x_i$ of $h_j$ may contribute its rank to the rank of input position $x_i$ of $f$, provided rank $\tau_j$ is not $\iota$, that is:
\[ f(x_1, \ldots, x_l) \text{ has ranks } \max(\text{smax}(\tau_1, \rho_1), \text{smax}(\tau_2, \delta_1)), \ldots, \max(\text{smax}(\tau_1, \rho_1), \text{smax}(\tau_2, \delta_1)) \]

In particular, if for example \( h_1 \) is a constant function, then input position \( x_i \) of \( f \) has rank \( \text{smax}(\tau_2, \delta_i) \).

- \( f \) is defined by \textit{primitive recursion}, say \( f(\vec{x}, 0) = g(\vec{x}) \) and \( f(\vec{x}, y + 1) = h(\vec{x}, y, f(\vec{x})) \). Assume that
  \[
  \begin{align*}
  g(x_1, \ldots, x_l) & \text{ has ranks } \tau_1, \ldots, \tau_l \\
  h(x_1, \ldots, x_l, u, v) & \text{ has ranks } \rho_1, \ldots, \rho_l, \rho_u, \rho_v.
  \end{align*}
  \]

If the rank \( \rho_v \) of the \textit{critical input position} \( v \) is \( q \), then \( f \) is called a \textit{recursion at rank} \( q \), and

\[ f(x_1, \ldots, x_l, y) \text{ has ranks } \max(\tau_1, \rho_1, q), \ldots, \max(\tau_l, \rho_l, q), \max(\rho_u, q + 1) \]

and we say:

- \( f \) is a \textit{top recursion} if \( q \geq \max\{\tau_1, \rho_1, \ldots, \tau_l, \rho_l, \rho_u\} \)
- \( f \) is a \textit{side recursion} otherwise

Otherwise if the \textit{critical} input position \( v \) is redundant, then \( f \) is called \textit{flat recursion}, and

\[ f(x_1, \ldots, x_l, y) \text{ has ranks } \max(\tau_1, \rho_1), \ldots, \max(\tau_l, \rho_l) \cup R(\rho_u). \]

In either case of primitive recursion, each position \( x_i \) of \( g, h \) may “contribute” to the rank of \textit{parameter position} \( x_i \) of \( f \), but for recursions at rank \( q \), this rank must be lifted to the rank \( q \) of the critical input position. For otherwise one could define functions of overall rank, \( k \) say, with rates of growth exceeding that available at Grzegorczyk level \( k+1 \). To see this, first inspect the examples above in the light of “ranks”:

The definition of addition \( \text{add}(x, y) \) is a top recursion and has ranks 1, 0.

The definition of multiplication \( \text{mult}(x, y) \) is a side recursion and has ranks 1, 1. For \( \text{mult} \) is \( R(g, h) \) with \( g(y) = 0 \) and \( h(y, u, v) = \text{add}(y, v) \), thus by “contribution” \( h(y, u, v) \) has ranks 1, \( \iota \), 0.

The definition of exponentiation \( \text{exp} \) is a top recursion and has rank 2, because \( \text{exp} \) is \( R(g, h) \) with \( g(y) = S(0) \) and \( h(y, u, v) = \text{add}(v, v) \), thus by “contribution” \( h(y, u, v) \) has ranks \( \iota, \iota, 1 \).

Now consider functions \( g, h \) defined by \( g(x) := x \) and \( h(x, u, v) := \text{mult}(v, v) = v^2 \). Then \( g \) has rank 0, and \( h \) has ranks \( \iota, \iota, 1 \). We obtain that function \( f \) defined by primitive recursion from \( g, h \), that is,

\[
\begin{align*}
  f(x, 0) &= x \\
  f(x, y + 1) &= h(x, y, f(x, y)) = f(x, y)^2
\end{align*}
\]

satisfies \( f(x, y) = x^{2^y} \). If the rank of parameter \( x \) were not lifted to the rank of the critical input position, that is, to 1, then \( f(x, y) \) would have ranks 0, 2. Then by “contribution” the function \( h' \) defined by

\[ h'(x, u, v) := f(v, S(0)) = v^2 \]

would have rank \( \iota, \iota, 0 \). Accordingly, the exponential function \( f \) could be defined by primitive recursion from \( g, h' \) with ranks \( \iota, \iota, 1 \). So we see that for a definition by recursion, one cannot dispense with lifting the rank of a parameter position to the rank of the critical input position.
Flat recursions just use the built-in mechanism for decrementing the recursion argument or for case analysis, whether the recursion argument is zero (or even in case of recursion on notation), such as predecessor and conditional. The present approach accounts for redundant input positions, thus predecessor \( P(x) \) has rank 0, and conditional \( C(x, y, z) \) has ranks 0, 0, 0. This ability is useful for various reasons: Firstly, it helps to keep ranks as low as one can hope for; secondly, it allows programs to be optimised before run time; thirdly, no initial functions other than zero and successor(s) are required.

3.2. The term systems \( \mathbb{PR}_1 \) and \( \mathbb{PR}_2 \)

Terms in \( \mathbb{PR}_1 \) and \( \mathbb{PR}_2 \) use only ground type variables \( x, y, z, u, v \), possibly with subscripts, and ground type recursion. Thus, types are the ground type \( \iota \), and if \( \tau \) is a type, then so is \( \iota \rightarrow \tau \). In other words, the objects under consideration are just functions over the natural numbers. Terms in \( \mathbb{PR}_i \), with their types, are:

\[
\begin{align*}
\mathbb{PR}_1 & := x \mid (\lambda x.r^\tau)^{\iota\rightarrow\tau} \mid (r^{\iota\rightarrow\tau} s^\tau) \mid 0, S \mid (R\ g^\iota h^{|\iota|\iota})^{\iota\rightarrow\iota} \\
\mathbb{PR}_2 & := x \mid (\lambda x.r^\tau)^{\iota\rightarrow\tau} \mid (r^{\iota\rightarrow\tau} s^\tau) \mid 0, s_0, s_1 \mid (R\ N\ g^\iota h^{|\iota|\iota} h^{|\iota|\iota})^{\iota\rightarrow\iota}
\end{align*}
\]

Observe that the recurser \( \mathbb{RN} \) follows the usual definition of recursion on notation (cf. Background 2.4), requiring two step terms \( h_0, h_1 \). Recall that a binary numeral is either 0 or \( s_{i_1} \ldots s_{i_k} 0 \) where \( i_j \in \{0, 1\} \), and that \( s_{i_j} \) denote binary numerals distinct from 0. The conversion rules for \( \mathbb{RN} \) are then as expected:

\[
\begin{align*}
\mathbb{RN} g \ h_0 \ h_1 \ 0 \ & \rightarrow \ g \\
\mathbb{RN} g \ h_0 \ h_1 \ (s_{i_j} \ n) & \rightarrow \ h_i \ n \ (\mathbb{RN} g \ h_0 \ h_1 \ n)
\end{align*}
\]

3.3. The \( \mu \)-measure on \( \mathbb{PR}_1 \) and \( \mathbb{PR}_2 \)

To determine the \( \mu \)-measure of a term \( t \), we assign ranks \( n \in \mathbb{N} \) to some of the free occurrences of variables in \( t \). Intuitively, the rank of a free variable \( x, n \) say, provides the information that \( x \) controls \( n \) top recursions. To preserve this information after \( \lambda \)-abstraction \( x \), we also consider ranked types. They result from simple types by decorating some input positions with a rank \( n \). In that way we bookkeep all ranked free variables and ranked input positions in a term so as to follow up the “control” in \( t \).

A ranked variable is a pair \( (x, n) \) with \( n \in \mathbb{N} \). We use capital Latin letters \( X, Y, Z \), possibly with subscripts, to denote finite sets \( \{(x_1, n_1), \ldots, (x_l, n_l)\} \) of ranked variables.

Ranked types are \( \iota \), and if \( \sigma \) is a ranked type, then so are \( (\iota \rightarrow \sigma) \) and \( (n \rightarrow \sigma) \) where \( n \in \mathbb{N} \). We use \( \tau \) and \( \rho \), possibly with subscripts, to denote \( \iota \) or any \( n \in \mathbb{N} \).

Furthermore, we use the following abbreviations:

\[
\begin{align*}
\text{VAR}(X) & := \{x \mid \exists k: (x, k) \in X\} \quad \text{variables of } X \\
X - x & := \{(y, j) \in X \mid y \neq x\} \quad \text{\( X \) minus \( x \)-part} \\
\text{RANK}(x, X) & := \max\{k \mid (x, k) \in X\} \quad \text{rank of } x \text{ in } X \\
X_{[l]} & := \{(y, \max(j, l)) \mid (y, j) \in X\} \quad \text{lifting of } X \text{ to } l \\
R(\tau) & := \begin{cases} l & \text{if } \tau = l \\ 0 & \text{else} \end{cases}
\end{align*}
\]

We define \( t \) has ranked free variables \( X \) and ranked type \( \sigma \), denoted \( \mu(t) = X; \sigma \), by induction on the structure of \( t \in \mathbb{PR}_i \).
Definition 3.3.1 (The $\mu$-measure). $\mu(t) = X; \sigma$ is inductively defined as follows:

(i) $\mu(x) := \{(x, 0)\}; \iota$

(ii) $\mu(0) := \emptyset; \iota$, and $\mu(c) := \emptyset; (0 \to \iota)$ for $c$ among $S, s_0, s_1$.

(iii) If $\mu(r) = X; \sigma$ then

$$\mu(\lambda x. r) := \begin{cases} X \cdot x; (\text{RANK}(x, X) \to \sigma) & \text{if } x \in \text{VAR}(X) \\ X; (\iota \to \sigma) & \text{else.} \end{cases}$$

(iv) If $\mu(r) = X; (\tau \to \sigma)$ and $\mu(s) = Y; \iota$ then

$$\mu(rs) := \begin{cases} X \cup Y; \tau; \sigma & \text{if } \tau = \iota \\ X; \sigma & \text{else.} \end{cases}$$

(v) If $\mu(r) = X; \iota$ and $\mu(s) = Y; (\tau_0, \tau_1 \to \iota)$ then

$$\mu(\text{R }rs) := X_{\lbrack l \rbrack} \cup Y_{\lbrack l \rbrack}; (k \to \iota)$$

where $l := \text{R}(\tau_1)$ and $k := \max\{\text{R}(\tau_0), 1 + l\}$ if $\tau_1 = \iota$, otherwise $k := \text{R}(\tau_0)$.

(vi) If $\mu(r) = X; \iota$ and $\mu(t_0) = Y; (\tau_0, \tau_1 \to \iota)$ and $\mu(t_1) = Z; (\rho_0, \rho_1 \to \iota)$, then

$$\mu(\text{RN }rt_0 t_1) := X_{\lbrack l \rbrack} \cup Y_{\lbrack l \rbrack} \cup Z_{\lbrack l \rbrack}; (k \to \iota)$$

where $l := \max\{\text{R}(\tau_1), \text{R}(\rho_1)\}$ and $k := \max\{\text{R}(\tau_0), \text{R}(\rho_0), 1 + l\}$ if $\tau_1 \neq \iota$ or $\rho_1 \neq \iota$, otherwise $k := \max\{\text{R}(\tau_0), \text{R}(\rho_0)\}$.

We say $t$ has $\mu$-measure $n$, denoted $\mu(t) = n$, if

$$\mu(t) = X; \sigma \text{ and } n = \max\{k \mid \exists x: (x, k) \in X\} \cup \{l \mid l \in \mathbb{N}, l \text{ occurs in } \sigma\}.$$  

One can read from the definition that $\mu(t) = X; \sigma$ implies $\text{VAR}(X) \subseteq \text{FV}(t)$. Thus, if $t$ is closed, then $\mu(t) = \emptyset; \sigma$, and in this case we write $\mu(t) = \sigma$ for short.

Note that (i) sets the stage for $\mu$ by initialising free variables with value 0. This initial value may change or vanish according to the remaining rules.

Definition 3.3.2. Let $t$ be any recursion $\text{R }rs$ (and similar for $\text{RN }rt_0 t_1$).

- $t$ is a recursion at rank $l$ if $\mu(s) = Y; (\tau_0, l \to \iota)$.
- $t$ is a top recursion if $t$ is a recursion at rank $\max\{\mu(r), \mu(s)\}$. Otherwise $t$ is called side recursion.
- $t$ is called flat recursion if $\mu(s) = Y; (\tau_0, \iota \to \iota)$. 
It turns out that only top recursions can cause an explosion in computational complexity while side recursions (and flat recursions) do not.

**Note 1.** Predecessor $P$, conditional $C$, and its binary versions $p, c$ all are defined by flat recursions:

\[
\begin{align*}
P &:= R \text{0} (\lambda uv. u) & p &:= RN \text{0} (\lambda uv. u) \\
C &:= \lambda xyz.R (\lambda uv. z) x & c &:= \lambda xyz.RN (\lambda uv. z) (\lambda uv.z) x \\
\end{align*}
\]

First we will clarify the role of non-ranked input positions or variables. Non-ranked free variables only occur in subterms without computational meaning to the algorithm, thus input positions of type $\iota$ are redundant (cf. [44]).

**Definition 3.3.3.** For a function $f : \mathbb{N}^k \rightarrow \mathbb{N}$ an input position $j$ of $f$, that is, $1 \leq j \leq k$, is called redundant if $f(\tilde{a}, b, \tilde{c}) = f(\tilde{a}, b', \tilde{c})$ for all $\tilde{a} := a_1, \ldots, a_{j-1}, b \neq b'$ and $\tilde{c} := c_{j+1}, \ldots, c_k$.

**Lemma 3.3.4 (Redundancy).** For all terms $t \in PR_1$ and lists of variables $\vec{x} := x_1, \ldots, x_p \supseteq \text{FV}(t)$ with $\mu(\lambda \vec{x}. t) = (\tau_1, \ldots, \tau_p, \tau_{p+1}, \ldots, \tau_q \rightarrow \iota)$, each input position $j$ of $[\lambda \vec{x}. t]$ with $\tau_j = \iota$ is redundant.

**Proof.** If $t$ is a variable $y$, then every input position $j$ of $[\lambda \vec{x}. t]$ where $x_j \neq y$ is redundant. If $t$ is a constant, then every input position $j$ of $[\lambda \vec{x}. t]$ with $1 \leq j \leq p$ is redundant. If $t$ is $\lambda y.r$, then the claim for $t$ follows from the induction hypothesis on $r$ with respect to $\vec{x}, y \supseteq \text{FV}(r)$. So consider the case where $t$ is $rs$, assuming $\mu(r) = X; (\tau, \tau_{p+1}, \ldots, \tau_q \rightarrow \iota)$ and $\mu(s) = Y; \iota$. First note that

\[
(*) \quad [\lambda \vec{x}. rs](\vec{a}, \vec{b}) = [\lambda \vec{x}. r](\vec{a}, [\lambda \vec{x}. s](\vec{a}), \vec{b}).
\]

**Subcase $\tau = l$.** Hence $\mu(t) = X \cup Y_1; (\tau_{p+1}, \ldots, \tau_q \rightarrow \iota)$. For every input position $j > p$ of $[\lambda \vec{x}. t]$, the claim follows from the induction hypothesis on $r$ for $j + 1$. For $1 \leq j \leq p$ with $\tau_j = \iota$ we conclude $x_j \notin \text{VAR}(X \cup Y_1)$ which by $(*)$ and the induction hypothesis on $r, s$ implies that $j$ is a redundant input position of $[\lambda \vec{x}. t]$.

**Subcase $\tau = \iota$.** Hence $\mu(t) = X; (\tau_{p+1}, \ldots, \tau_q \rightarrow \iota)$ and $[\lambda \vec{x}. t](\vec{a}, \vec{b}) = [\lambda \vec{x}. r](\vec{a}, 0, \vec{b})$ by the induction hypothesis on $r$ and $\iota$. Thus, the claim for $t$ follows from the induction hypothesis on $r$.

Finally, consider the case where $t$ is a recursion $R \text{r s}$ (and similar for $RN \text{r h_0 h_1}$) with $\mu(r) = X; \iota$ and $\mu(s) = Y; (\tau_0, \tau_1 \rightarrow \iota)$. Recall that $\mu(R \text{r s}) = X_1 \cup Y_1; (k \rightarrow \iota)$ for $l$ and $k$ as defined above, and that $\mu(\lambda \vec{x}. t) = (\tau_1, \ldots, \tau_p, k \rightarrow \iota)$. Hence for every $\tau_j = \iota$ we conclude $x_j \notin \text{VAR}(X_1 \cup Y_1)$ which by the induction hypothesis on $r, s$ implies that $j$ is a redundant input position of $[\lambda \vec{x}. t]$. \qed

**Corollary 3.3.5 ($\mu$-Constants).** Every term $t$ with $\mu(t) = \emptyset; \iota$ denotes a constant $c$, that is, $[t]_\emptyset = c$ for every environment. In that case we have $[t[\vec{0}/\text{FV}(t)]] = c$, and every reduction sequence for $t[\vec{0}/\text{FV}(t)]$ ends with the numeral $S^0$ (or with a binary numeral denoting $c$ in the case of $t \in PR_2$).

**Proof.** By Redundancy $[\lambda \text{FV}(t).t]$ is a constant function, implying the statement of the corollary. \qed

In contrast to the approach in [46], [45], the current definition of the measure $\mu$ admits that the $\mu$-measure of a subterm may dominate that of the whole term. Thus we would run into a problem when trying to prove by induction on the structure of terms that e.g. every closed term $t \in PR_1$ with $\mu$-measure $n$ defines a function in Grzegorczyk class $E_{n+1}$. 

There are two forms of subterms whose \( \mu \)-measure dominates that of the overall term. One is caused by redundant applications \( rs \) where \( \mu(r) = X; (l \to \sigma) \) and \( \mu(r) < \mu(s) \). By Redundancy (3.3.4) one can replace \( s \) with 0. The other form could be dubbed constant-depth recursion, that is, applications \( rs \) where \( \mu(r) = X; (l \to \sigma) \), \( \mu(r<s) < l \) and \( \mu(s) = \emptyset ; \mu \). By Corollary 3.3.5 \( s \) denotes a constant that is eventually passed to a “recursion position” of \( r \). For example, consider the definition of addition

\[
+ := \lambda xy. \mathbf{R} y (\lambda uv. \mathbf{S} v) x
\]

satisfying \( \mu(+) = (1, 0 \to \iota) \). Hence \( \mu(+(\mathbf{SS}0)) = (0 \to \iota) \).

In order to eliminate constant-depth recursions without increasing the \( \mu \)-measure, we first show that the \( \mu \)-measure is stable under \( \beta, \eta \) reductions. Essentially we need to know how \( \mu \) behaves when substituting a term \( s \) with \( \mu(s) = Y; \iota \) for a free variable \( x \) of a term \( t \) with \( \mu(t) = X; \sigma \). In case of \( x \in \text{VAR}(X) \), the central technical term is

\[
Y_{\iota}(x, X) := \{ (y, \max(j, k) | (y, j) \in Y, (x, k) \in X \}
\]

the idea being that in \( t[s/x] \) every free occurrence of \( x \) with a rank \( k \) will be replaced with \( s \) so that the ranks of \( y \in \text{VAR}(Y) \) must be lifted to rank \( k \).

**Lemma 3.3.6 (Substitution).** Suppose that \( \mu(t) = X; \sigma \) and \( \mu(s) = Y; \iota \), with \( s \) substitutable for \( x \) in \( t \).

(a) If \( x \in \text{VAR}(X) \) then \( \mu(t[s/x]) = X-x \cup Y_{\iota}(x, X); \sigma \).

(b) If \( x \notin \text{VAR}(X) \) then \( \mu(t[s/x]) = X; \sigma \).

**Proof.** As part (b) follows from a straightforward induction on the structure of \( t \), we focus on the proof of part (a). We use the following properties of “lifting” which can be easily verified for arbitrary finite sets \( U, V, Y \) of ranked variables and \( l \in \mathbb{N} \):

1. \( Y_{\iota}(x, U \cup V) = Y_{\iota}(x, U) \cup Y_{\iota}(x, V) \)
2. If \( x \notin \text{VAR}(U) \) then \( Y_{\iota}(x, U) = \emptyset \).
3. \( (Y_{\iota}(x, U))_{\iota l} = Y_{\iota}(x, U_{\iota l}) \)
4. \( (U-x)_{\iota l} = U_{\iota l-x} \)
5. \( (U \cup V)_{\iota l} = U_{\iota l} \cup V_{\iota l} \)
6. \( (U_{\iota l})_{\iota l} = U_{\iota l} \)
7. \( U_{\iota 0} = U \)

We proceed by induction on the structure of \( t \) showing part (a), where we assume \( x \in \text{VAR}(X) \) for each case \( \mu(t) = X; \sigma \). If \( t \) is \( \iota \), then \( \mu(t) = \{ (x, 0) \}; \iota \) and \( t[s/x] = s \), and so we are done by (7).

**Case** \( t \) is \( \lambda y.r \) where \( \mu(r) = U; \rho \). As \( x \) is among the ranked free variables of \( t \), we know \( x \neq y \), and as \( s \) is substitutable for \( x \) in \( t \), we conclude \( y \notin \text{FV}(s) \supseteq \text{VAR}(Y) \) and thus \( y \notin \text{VAR}(Y_{\iota}(x, U)) \).

**Subcase** \( y \in \text{VAR}(U) \), hence \( \mu(t) = \mathbf{U}-y; (\text{RANK}(y, U) \to \rho) \). As \( x \in \text{VAR}(U) \), the induction hypothesis on \( r \) implies \( \mu(t[s/x]) = (\mathbf{U}-x \cup Y_{\iota}(x, U))_{\iota-y}; (\text{RANK}(y, \mathbf{U}-x \cup Y_{\iota}(x, U)) \to \rho) \). Since \( y \notin \text{VAR}(Y_{\iota}(x, U)) \) and \( x \neq y \), we conclude \( \mu(t[s/x]) = (\mathbf{U}-y)_{\iota-x} \cup Y_{\iota}(x, \mathbf{U}-y); (\text{RANK}(y, \mathbf{U}) \to \rho) \).
Subcase \( y \notin \text{VAR}(U) \), thus \( \mu(t) = U; (t \rightarrow \rho) \). As \( x \in \text{VAR}(U) \) and \( y \notin \text{VAR}(U \cup Y_{1}(x,U)) \), the induction hypothesis on \( r \) implies \( \mu(t[s/x]) = U \setminus x \cup Y_{1}(x,U) \cup (\iota \rightarrow \rho) \), concluding the current case.

Case \( t \) is \( r_{1}r_{2} \) where \( \mu(r_{1}) = U; (t \rightarrow \sigma) \) and \( \mu(r_{2}) = V; \iota \). If \( \iota = \iota \) then \( \mu(t) = U; \sigma \) and \( x \in \text{VAR}(U) \), so the induction hypothesis on \( r_{1} \) implies \( \mu(t[s/x]) = U \setminus x \cup Y_{1}(x,U) \cup (l \rightarrow \sigma) \). Hence \( \mu(t) = U \cup V_{l} \cup \sigma \) and we must show \( \mu(t[s/x]) = (U \cup V_{l}) \cup Y_{1}(x,U) \cup (l \rightarrow \sigma) \).

Subcase \( x \in \text{VAR}(U) \setminus \text{VAR}(V) \). By (1), (2) we must show \((*)\) \( \mu(t[s/x]) = U \cup (V_{l} \setminus x) \cup Y_{1}(x,V_{l}); \sigma \). The induction hypothesis on \( r_{1} \) yields \( \mu(r_{1}[s/x]) = (U \setminus x) \cup Y_{1}(x,V_{l}); (l \rightarrow \sigma) \), and part (b) of the lemma gives \( \mu(r_{2}[s/x]) = V; \iota \). Now \((*)\) follows by definition.

Subcase \( x \in \text{VAR}(V) \setminus \text{VAR}(U) \). By (1), (2) we must show \((*)\) \( \mu(t[s/x]) = U \cup (V_{l} \setminus x) \cup Y_{1}(x,V_{l}); \sigma \). We conclude \( \mu(r_{1}[s/x]) = U; (l \rightarrow \sigma) \) from (b), and the induction hypothesis on \( r_{2} \) yields \( \mu(r_{2}[s/x]) = (V_{l} \setminus x) \cup Y_{1}(x,V_{l}); \iota \). Hence \( \mu(t[s/x]) = U \cup (V_{l} \setminus x) \cup Y_{1}(x,V_{l}) \cup (Y_{1}(x,V_{l}); \iota; \sigma) \) by definition and (5). Now \((*)\) follows from (4) and (3), (6).

Subcase \( x \in \text{VAR}(V) \cap \text{VAR}(U) \). We have to show \((*)\) \( \mu(t[s/x]) = (U \setminus x) \cup (V_{l} \setminus x) \cup Y_{1}(x,U) \cup Y_{1}(x,V_{l}); \sigma \) according to (1). The induction hypothesis yields \( \mu(r_{1}[s/x]) = (U \setminus x) \cup Y_{1}(x,U); (l \rightarrow \sigma) \) and \( \mu(r_{2}[s/x]) = (V_{l} \setminus x) \cup Y_{1}(x,V_{l}); \iota \). Hence \( \mu(t[s/x]) = (U \setminus x) \cup Y_{1}(x,U) \cup (V_{l} \setminus x) \cup Y_{1}(x,V_{l}); \iota; \sigma \) by definition and (5). Now \((*)\) follows from (4) and (3), concluding the current case.

Finally, if \( t \) is a recursion \( \mathbf{R} \beta \mathbf{R} \) (similar for recursion on notation), we argue component-wise, using the induction hypothesis and possibly part (b) of the lemma.

It goes without saying that the \( \mu \)-measure is invariant under renaming of bound variables. The previous lemma allows one to give a short proof of this natural requirement.

**Corollary 3.3.7 (Invariance under bound renaming).** If \( \mu(t) = X; \sigma \) and \( t' \) results from \( t \) by renaming bound variables, then \( \mu(t') = X; \sigma \).

**Proof.** It suffices to consider the case where \( t \) is \( \lambda y.r \) and \( t' \) is \( \lambda z.r[z/y] \) such that \( z \notin \text{FV}(r) \) and \( z \neq y \). Suppose that \( \mu(r) = U; \sigma \).

Subcase \( y \in \text{VAR}(U) \). Hence \( \mu(t) = U \setminus y; (\text{RANK}(y,U) \rightarrow \sigma) \) by definition, and Substitution (3.3.6) (a) yields \( \mu(r[z/y]) = U \setminus y \cup \{(z,0)\}_{1(y,U)}; \sigma \). Now, since \( \{(z,0)\}_{1(y,U)} = \{(z,k) \mid (y,k) \in U\} \), we conclude \( \mu(t') = U \setminus y; (\text{RANK}(y,U) \rightarrow \sigma) \).

Subcase \( y \notin \text{VAR}(U) \). Hence \( \mu(t) = U; (\iota \rightarrow \sigma) \) and \( \mu(r[z/y]) = U; \sigma \) by Substitution (3.3.6) (b). As \( z \notin \text{FV}(r) \supseteq \text{VAR}(U) \), we conclude \( \mu(t') = U; (\iota \rightarrow \sigma) \).

Henceforth, in considerations on \( \mu \) involving substitution, we can always assume that \( s \) is substitutable for \( x \) in \( t \) when writing \( t[s/x] \).

We write \( t \rightarrow_{\beta} t' \) if \( t \) reduces to \( t' \) using any conversion rules other than \( \mathbf{R} \) or \( \mathbf{R} \mathbf{N} \)-conversions. For obvious reasons, one cannot expect that \( t \rightarrow_{\beta} t' \) with \( \mu(t) = X; \sigma \) implies \( \mu(t') = X; \sigma \). What we can expect is that \( \mu(t') = X'; \sigma \) such that \( \text{VAR}(X') = \text{VAR}(X) \) and \( \text{RANK}(x,X') = \text{RANK}(x,X) \) for all \( x \in \text{VAR}(X) \).

**Theorem 3.3.8 (Stability).** If \( t \rightarrow_{\beta} t' \) and \( \mu(t) = X; \sigma \), then \( \mu(t') = X'; \sigma \) such that \( \text{VAR}(X') = \text{VAR}(X) \) and \( \text{RANK}(x,X') = \text{RANK}(x,X) \) for all \( x \in \text{VAR}(X) \). Thus, if \( t \rightarrow_{\beta} t' \) then \( \mu(t') = \mu(t) \).

**Proof.** We proceed by induction on \( t \rightarrow_{\beta} t' \). The statement is obvious if \( t \) is \( s_{0}t \) and \( t' = 0 \).

If \( t \) is \( \lambda x.r \), then \( t' \) is \( \lambda x.r' \) and \( r \rightarrow_{\beta} r' \). Hence we are done by the induction hypothesis on \( r \).
If \( t = rs \) and \( t' = r's \) with \( r \rightarrow^* r' \), or \( t' = rs' \) with \( s \rightarrow^* s' \), then the statement for \( t \) follows from the induction hypothesis on the reduced component.

Suppose that \( t \) is \((\lambda x.r)s\), and that \( t' \) is \( r[s/x] \) where \( \mu(r) = X;\sigma \) and \( \mu(s) = Y;\iota \). If \( x \notin \text{VAR}(X) \) then \( \mu(t) = X;\sigma \) by definition, and \( \mu(t') = X;\sigma \) by Substitution (3.3.6) (b). Otherwise if \( x \in \text{VAR}(X) \) then \( \mu(t) = X-x \cup \text{VAR}(x);\sigma \) by definition, and \( \mu(t') = X-x \cup \text{VAR}(x);\sigma \) by Substitution (3.3.6) (a). Now observe that \( x \in \text{VAR}(X) \) implies \( \text{VAR}(Y_\uparrow(x,X)) = \text{VAR}(Y_{\text{RANK}}(x,X)) \) and \( \text{RANK}(y,Y_\uparrow(x,X)) = \max\{\text{RANK}(y,Y),\text{RANK}(x,X)\} = \text{RANK}(y,Y_{\text{RANK}}(x,X)) \) for all \( y \in \text{VAR}(Y_\uparrow(x,X)) \).

Finally, if \( t \) is \( R\cdot r\cdot s \) (similar for \( RN\cdot r\cdot h_0\cdot h_1 \)), then \( t' \) results from \( t \) by reducing one of the components \( r \) or \( s \). Thus, the claim follows from the induction hypothesis of the reduced component.

While Stability (3.3.8) guarantees that the \( \mu \)-measure is preserved when performing \( \beta \) reductions, unfolding constant-depth recursions \( t := R\cdot g\cdot h \cdot n \) can decrease the \( \mu \)-measure for various reasons. To see this, suppose that \( \mu(g) = X;\iota \) and \( \mu(h) = Y;\iota \). Then one obtains
\[
\mu(t) = X_\uparrow \cup Y_\uparrow;\iota \quad \text{where} \quad l = R(\iota).
\]

Obviously, if \( n = 0 \) then \( t \rightarrow^* g \) and \( \mu(g) < \mu(t) \) might happen. So assume that \( n \) denotes a non-zero number, hence \( t \rightarrow^* t' := h \cdot n - 1 \ (... \cdot (h \cdot 0 \cdot g) \ldots ) \). In that case there are two possibilities for \( \mu(t') < \mu(t) \).

One is when \( R\cdot g\cdot h \) is a flat recursion \( (\tau_1 = \iota) \), because then \( \mu(t') = Y;\iota \). The other one is when \( R\cdot g\cdot h \) is a recursion at rank \( l \) \((\tau_1 = l) \) and \( n \) is \( \text{SO} \); in this case \( t' \) is \( h \cdot 0 \cdot g \) with \( \mu(t') = Y \cup X_\uparrow;\iota \), and so \( \mu(t') < \mu(t) \) might happen for \( X = 0 \). Observe that if \( n \) denotes a number \( \geq 2 \), then \( \mu(t') = X_\uparrow \cup Y_\uparrow;\iota \).

It follows that if we want to eliminate constant-depth recursions, then we will have to unfold recursions and that will decrease the overall \( \mu \)-measure.

**Definition 3.3.9.** Let \( t \) be any term.

- \( t \) is called well-structured if every subterm \( s \) of \( t \) satisfies \( \mu(s) \leq \mu(t) \).
- \( t \) is called redundancy-free if \( t \) has no subterm \( rs \) such that \( \mu(r) = X;\iota \rightarrow\sigma \).
- \( t \) is called fair if \( t \) is well-structured and redundancy-free.

**Lemma 3.3.10 (Fairness).** Every term \( t \) has a fair term \( t' \) equivalent to \( t \) such that \( \mu(t') \leq \mu(t) \).

**Proof.** By Stability (3.3.8) and the observation above one obtains a fair term equivalent to \( t \) by computing the normal form \( nf(t) \) of \( t \).

The cost of computing the normal form \( nf(t) \) might be extremely high, because one already might need super-exponential time (in the size of \( t \)) to compute the \( \beta \)-normal form. But to obtain fair terms, one need not to fully normalise \( t \) but convert only some \( \beta \)-redexes and unfold only some constant-depth recursions.

### 3.4. The Bounding Theorem

The Bounding Theorem states that every function represented by a term in \( PR_i \) with \( \mu \)-measure \( n \) can be bounded by a function in \( \mathcal{E}_{n+1} \). Once this is established, the characterisation theorems of section 5 follow from straightforward inductive arguments, because then every recursion in \( PR_i \) with \( \mu \)-measure \( n \) can be
turned into a bounded recursion in $E_{n+1}$. In fact, as side recursions do not increase the $\mu$-measure, the hard work is to show the Bounding Theorem (cf. [45]).

As for the bounding functions, we use so-called Grzegorczyk polynomials. These functions have remarkable properties which reflect the power and flexibility of the interplay of top and side recursions. Similar to ordinary polynomials, Grzegorczyk polynomials are built from generalised addition $+^{r}$ and multiplication $\times^{r}$ functions, defined simultaneously with generalised successor functions $S^{r}$.

**Definition 3.4.1.** For every $r \geq 1$, the generalised successor, addition and multiplication function, denoted $S^{r}, +^{r}, \times^{r}$ respectively, are inductively defined as follows, where we give both recursion equations and representations in PR$_{1}$:

$$S^{1}(x) := x + 1 \quad \quad S^{1} := S$$

$$0 +^{r} y := y \quad \quad (x + 1) +^{r} y := S^{r}(x +^{r} y) \quad \quad +^{r} := \lambda xy. R \ y (\lambda uv. S^{r} v) \ x$$

$$0 \times^{r} y := 0 \quad \quad (x + 1) \times^{r} y := y +^{r} (x \times^{r} y) \quad \quad \times^{r} := \lambda xy. R \ 0 (\lambda uv. +^{r} yv) \ x$$

$$S^{r+1}(y) := S^{r}((y +^{r} y) \times^{r} (y +^{r} y)) \quad \quad S^{r+1} := \lambda y. S^{r} (\times^{r}(y +^{r} y) (y +^{r} y))$$

Observe that $+^{r}$ is defined by a top recursion from $S^{r}$, while $\times^{r}$ is defined by a side recursion from $+^{r}$, and $S^{r+1}$ is defined explicitly from $S^{r}, +^{r}, \times^{r}$. Therefore one obtains inductively:

$$\mu(S^{r}) = (r \to 1)$$

$$\mu(+^{r}) = (r, r \to 1)$$

$$\mu(\times^{r}) = (r, r \to 1)$$

For legibility, we often write $+^{r}$ and $\times^{r}$ for $[+^{r}], [\times^{r}]$ respectively.

**Lemma 3.4.2 (Monotonicity).** Let $r, s \geq 1$ and $x, y, z$ be arbitrary natural numbers.

(a) $x +^{r} y = S^{r}(x)(y)$

(b) $x \times^{r} y = (x \cdot y) +^{r} 0$

(c) $+^{1}$ is the ordinary addition function and $\times^{1}$ the ordinary multiplication function.

(d) The functions $S^{r}, +^{r}, \times^{r}$ are strongly monotonic, that is, e.g. for $+^{r}$, if $x \leq x', y \leq y', r \leq s$, then $x +^{r} y \leq x' +^{s} y'$. In particular, $y < S^{r}(y)$, $\max(x, y) \leq x +^{r} y$ and $x \cdot y \leq x \times^{r} y$.

(e) If $r \leq s$ then $x +^{r} (y +^{s} z) \leq (x +^{r} y) +^{s} z$.

(f) $y \circ y \leq S^{r+1}(y)$ for $\circ \in \{+, \times, \max\}$ with $s \leq r$.

*Proof.* Part (a) is obvious from the recursion equations for $+^{r}$. Part (b) follows from (a) by a straightforward induction on $x$. Clearly, (a) and (b) imply (c). As for the first part of (d), by (b) and definition of $S^{r}$, it suffices to prove strongly monotonicity for $+^{r}$. We proceed by induction on $s \geq 1$, where the base case is obvious from (c). As for the step case $s \to s + 1$, we proceed by side induction on $x'$, where the base case holds by
definition of \( +^{s+1} \). For the step case \( x' \rightarrow x' + 1 \), assume that \( r \leq s + 1, x \leq x' + 1 \) and \( y \leq y' \). First observe that the main induction hypothesis implies \( (*) \) \( S^{s+1}(z') \geq S^m(z) \) for all \( s + 1 \geq m \) and \( z' \geq z \). In particular, \( (*) \) and \( (a) \) imply \( y < S(y) \leq S^{s+1}(x' + 1)(y) = (x' + 1) +^{s+1} y \). Thus, it suffices to consider the case where \( x \) is a successor \( x'' + 1 \). In that case we conclude the first part of \( (d) \) as follows:

\[
(x' + 1) +^{s+1} y' = S^{s+1}(x' +^{s+1} y') \quad \text{by definition}
\]

\[
\geq S^r(x'' +^r y) \quad \text{by \( (*) \) and the side I.H.}
\]

\[
= x +^r y \quad \text{by definition}
\]

As \( S^r \) is strongly monotonic, we obtain \( y < S(y) \leq S^r(y) \). This and \( (a) \) imply \( x +^r y = S^r(x)(y) \geq \max(x, y) \), and hence \( x \times^r y \geq x \cdot y \) by \( (b) \). Part \( (e) \) follows from \( (d),(a) \), for if \( r \leq s \), we obtain:

\[
x +^r (y +^s z) \leq x +^s (y +^s z) \quad \text{by \( (d) \)}
\]

\[
= S^{s(x+y)}(z) \quad \text{by \( (a) \), applied two times}
\]

\[
\leq S^{s(x^r+y)}(z) \quad \text{by \( (d) \)}
\]

\[
= (x +^r y) +^s z \quad \text{by \( (a) \)}
\]

Part \( (f) \) follows from \( (d) \), for if \( s \leq r \), then \( y \times^s y \leq S^r((y +^r y) \times^r (y +^r y)) = S^{r+1}(y) \), and furthermore, \( y +^s y \leq S^r((y +^r y)^2) \leq S^r((y +^r y) \times^r (y +^r y)) \leq S^{r+1}(y) \). \( \square \)

Observe that \( +^r \) is neither commutative nor associative for \( r \geq 2 \). Thus, the “semi-associativity” stated in \( (d) \) is all one can hope for, but that together with the monotonicity properties stated in \( (c) \) prove sufficient for establishing all crucial estimations below.

**Definition 3.4.3.** Grzegorczyk polynomials \( q \) are inductively built up by

\[
q := c \mid x \mid \max(q, q) \mid q \circ q
\]

where \( \circ = \times^m, +^n \). For uniformity of notation, we sometimes allow \( \circ \) to be \( \max \).

For Grzegorczyk polynomials \( q \) we write \( q(\bar{x}) \) to mean \( \text{Var}(q) \subseteq \{ \bar{x} \} \). For arguments \( \bar{a} \), the value \( q(\bar{a}) \) of applying \( q \) to \( \bar{a} \) is defined as usual, that is, \( c(\bar{a}) := c, x_i(\bar{a}) := a_i \) and \( (q_1 \circ q_2)(\bar{a}) := q_1(\bar{a}) \circ q_2(\bar{a}) \).

We want to prove that every function defined by a term \( t \) with \( \mu \)-measure \( n \) can be bounded by a Grzegorczyk polynomial \( q_t \) in \( \mathcal{E}_{n+1} \). To make the construction work, however, we will have to strengthen the statement to the requirement that the “input ranks” of \( t \) coincide with those of \( q_t \). But as bounding involves the use of \( \max \), we would run into a problem, because every definition of \( \max \) has at least \( \mu \)-measure 1.

To overcome this obstacle, we define a modified measure \( \nu \) for Grzegorczyk polynomials only. Intuitively, \( \nu \) is very much like \( \mu \), except considering \( \max \) to be an initial function. The Modified Bounding lemma then states that every function represented by a term \( t \) with \( \mu \)-measure \( n \) can be bounded by a Grzegorczyk polynomial \( q_t \) in \( \mathcal{E}_{n+1} \) such that the “input ranks” of \( t \) assigned by \( \mu \) coincide with the “input ranks” of \( q_t \) assigned by \( \nu \). In fact, this implies the Bounding Theorem.

Grzegorczyk polynomials can be viewed as expressions built from constants, variables and “functions symbols” \( \times^m, +^n, \max \). Using the facts about the \( \mu \)-measure of the principal functions \( +^r, \times^r \), we define first the modified measure of a variable \( x \) in \( q \), denoted \( \nu(q, x) \). As with \( \mu \), the modified measure of \( q \), denoted \( \nu(q) \), is the maximum of all these “input ranks” \( \nu(q, x) \).

**Definition 3.4.4.** For Grzegorczyk polynomials \( q \) and variables \( x \), the modified measure of \( x \) in \( q \), denoted by \( \nu(q, x) \), is inductively defined as follows.
The modified measure of \( q \), denoted \( \nu(q) \), is defined by 
\[
\nu(q) := \max\{\nu(q, x) \mid x \in \Var(q)\}.
\]

Note that \( x \not\in \Var(q) \) implies \( \nu(q, x) = 0 \). Before we are going to state and prove the Modified Bounding lemma, we first set up the most striking properties of Grzegorczyk polynomials w.r.t. \( \nu \).

**Lemma 3.4.5 (Definability).** Every Grzegorczyk polynomial \( q \) belongs to \( \mathcal{E}_{\nu(q)+1} \).

**Proof.** First we show by induction on \( r \geq 1 \) that (*) \( S^r \in \mathcal{E}_r \) and \( +^r, \times^r \in \mathcal{E}_{r+1} \). The base case follows from Lemma 3.4.2 (c). As for the step case, the induction hypothesis yields \( S^r \in \mathcal{E}_r \) and \( +^r, \times^r \in \mathcal{E}_{r+1} \). By definition this implies \( S^{r+1} \in \mathcal{E}_{r+1} \), and by Lemma 3.4.2 (b) it suffices to show \( +^{r+1} \in \mathcal{E}_{r+2} \). This follows from the well-known fact [54] that any function defined by primitive recursion from functions in \( \mathcal{E}_n \), \( n \geq 2 \), belongs to \( \mathcal{E}_{n+1} \).

Now consider a Grzegorczyk polynomial \( q \), and let \( r = \nu(q) \). We define \( q \) in \( \mathcal{E}_{r+1} \) by induction on the structure of \( q \). The statement is obvious for the cases where \( q \) is a constant or a variable. If \( q \) is \( q_1 +^s q_2 \) or \( q_1 \times^s q_2 \) with \( s \geq r + 1 \), then either \( q \) does not contain any variables (in which case we are done) or \( q = q_1 +^{r+1} q_2 \), and \( q_1 \) is equivalent to a constant \( c \) and \( \nu(q_2) \leq r \), for any other form would result in a polynomial of higher modified measure than \( r = \nu(q) \). In the latter case the statement follows from the induction hypothesis on \( q_2 \), (*) and Lemma 3.4.2 (a). Otherwise if \( q \) is \( q_1 +^s q_2 \) or \( q_1 \times^s q_2 \) with \( s \leq r \), or \( q \) is \( \max(q_1, q_2) \), then \( q \in \mathcal{E}_{r+1} \) follows from the induction hypothesis and (*).

**Lemma 3.4.6 (Domination).** For every Grzegorczyk polynomial \( q(\vec{x}) \) one can find a constant \( c_q \) satisfying \( q(\vec{x}) \leq c_q + \nu(q)+1 \max \Var(\vec{x}) \).

**Proof.** Induction on the structure of \( q \), using \( \vec{x} := \Var(q) \). If \( q \) is a constant \( c \) then \( c_q := c \) will do; if it is a variable, we can define \( c_q := 0 \). Otherwise \( q(\vec{x}) \) is \( q_1(\vec{x}_1) \circ q_2(\vec{x}_2) \) with \( \circ \in \{+^s, \times^s, \max \} \). The induction hypothesis provides suitable constants \( c_{q_1}, c_{q_2} \) for \( q_1, q_2 \) respectively. Let \( r = \nu(q) \). If \( \circ \in \{+^s, \times^s, \max \} \) with \( s \leq r \), then we define \( c := \max(c_{q_1}, c_{q_2}) \) and conclude this case by:
\[
q(\vec{x}) \leq (c +^{r+1} \max \vec{x}) \circ (c +^{r+1} \max \vec{x}) \quad \text{by the I.H. and (3.4.2) (d)}
\leq S^{r+1}(S^{r+1}(c)(\max \vec{x})) \quad \text{by (3.4.2) (f), (a)}
= (c + 1) +^{r+1} \max \vec{x} \quad \text{by (3.4.2) (a)}
\]
If \( \circ \) is \( +^s \) with \( s > r + 1 \), or if \( \circ \) is \( \times^s \) with \( s > r \), then \( q \) must be equivalent to a constant, and we are done. If \( s = r + 1 \) then \( q = c +^{r+1} q_2 \) and hence \( c_q := c +^{r+1} c_{q_2} \) will do.

**Definition 3.4.7.** For Grzegorczyk polynomials \( q(\vec{x}) \), let \( \vec{x}^q \) result from \( \vec{x} \) by cancelling those \( x_i \) with \( \nu(q, x_i) < \nu(q) \). Furthermore, let \( +^q \) stand for \( +^{\nu(q)} \), and \( \times^q \) for \( \times^{\nu(q)} \). By \( +^0 \) we mean \( +^{1} \).
Lemma 3.4.8 (Top Level Separation). For every Grzegorczyk polynomial \( q \) one can find a Grzegorczyk polynomial \( q' \) over \( \vec{x}^2 \) such that

- \( q(\vec{x}) \leq q'(\vec{x}^2) + 4 \max \vec{x}^* \) for some \( \vec{x}^* \subseteq \text{Var}(q) \setminus \vec{x}^2 \),
- \( \nu(q', x) = \nu(q) \) for every \( x \in \vec{x}^2 \).

Proof. Induction on the structure of \( q \). The cases where \( q \) is a constant or a variable are obvious, as \( \vec{x}^2 = \vec{x} \) and \( q(\vec{x}) = q + 0 \). So assume that \( q = q_1(\vec{x}_1) \circ q_2(\vec{x}_2) \) with \( \vec{x}_i := \text{Var}(q_i) \) for \( i = 1, 2 \).

Case \( \circ \) is max. If \( \nu(q_1) < \nu(q_2) \) then \( q_1(\vec{x}_1) \leq c + q \max \vec{x}_1 \) by Domination (3.4.6), and \( \vec{x}^2 = \vec{x}^* \). Using the induction hypothesis on \( q_2 \), we define \( q'(\vec{x}^2) := \max(c, q'_2(\vec{x}^2)) \) to obtain \( q(\vec{x}) \leq q'(\vec{x}^2) + 4 \max(\vec{x}_1 \cup \vec{x}^2) \) by Lemma 3.4.2 (d). Similarly for the case \( \nu(q_1) > \nu(q_2) \). Otherwise if \( \nu(q_1) = \nu(q_2) \) then we conclude \( q(\vec{x}) \leq q'(\vec{x}^2) + 4 \max(\vec{x}_1^* \cup \vec{x}^2) \) for \( q'(\vec{x}^2) := \max(q'_1(\vec{x}_1^2), q'_2(\vec{x}_2^2)) \), using the induction hypothesis on \( q_1, q_2 \) and Lemma 3.4.2 (d).

Case \( \circ \) is \( + \). If \( \nu(q_1) < \nu(q_2) \) then we conclude \( q_2(\vec{x}_2) \leq c + 4 \max \vec{x}_2 \) from Domination (3.4.6), and \( \vec{x}^2 = \vec{x}_1 \). We define \( q'(\vec{x}^2) := q_1(\vec{x}_1) + q \) to obtain \( q(\vec{x}) \leq q'(\vec{x}^2) + 4 \max \vec{x}_2 \) by Lemma 3.4.2 (d), (e). Otherwise if \( \nu(q_1) = \nu(q_2) \) then we obtain \( q(\vec{x}) \leq q'(\vec{x}^2) + 4 \max \vec{x}^2 \) for \( q'(\vec{x}^2) := q_1(\vec{x}_1) + q'_2(\vec{x}_2^2) \), by the I.H. on \( q_2 \) and Lemma 3.4.2 (d), (e).

Case \( \circ \) is \( \times \). Hence \( \vec{x}^2 = \vec{x}_1 \cup \vec{x}_2 \), and \( q' := q \) will do, since \( q \leq q + 4 \).

Case \( \circ \) is \( \circ \) or \( \times \) with \( r < \nu(q) \). Then by Domination (3.4.6) and Lemma 3.4.2 (d) we obtain \( y \circ z \leq c + q \max(y, z) \) for some constant \( c \). Hence \( q \leq c + q \max(q_1, q_2) \), and \( q' := q + \) will do, where \( q'' \) is the Grzegorczyk polynomial obtained as above for \( \max(q_1, q_2) \).

Remaining cases. If \( \circ \) is \( \circ \) with \( r > \nu(q) + 1 \), or if \( \circ \) is \( \circ \) with \( r > \nu(q) \), then \( q \) is equivalent to a constant. If \( \circ \) is \( \circ + 1 \) with \( r = \nu(q) \), then \( q \) is of the form \( q = q_1(\vec{x}_2) \) and \( \vec{x}^2 = \vec{x}_2 \). In that case we use the induction hypothesis on \( q_2 \) to define \( q := c + q_1(\vec{x}_2) \) and \( \vec{x}^* := \vec{x}_2 \).

Domination, Top Level Separation and Decomposition below provide an elegant insight in the proof of the Separation lemma below.

Lemma 3.4.9 (Composition). If \( q(x_1, \ldots, x_m) \) and \( q_1(\vec{y}), \ldots, q_m(\vec{y}) \) are Grzegorczyk polynomials, then so is \( q(\vec{y}) := q_1(\vec{y}), \ldots, q_m(\vec{y}) \) satisfying

\[ \nu(q(\vec{y}), y) = \max\{\nu(q, x_i), \nu(q_i, y) \mid x_i \in \text{Var}(q), y \in \text{Var}(q_i)\}. \]

Proof. Induction on the structure of \( q \).

Lemma 3.4.10 (Decomposition). For every natural number \( s \), to each Grzegorczyk polynomial \( q(\vec{x}) \) one can assign a list \( s \text{-dcp}(q) = q_0(\vec{x}, w_1, \ldots, w_k), q_1(\vec{x}), \ldots, q_k(\vec{x}) \) of Grzegorczyk polynomials, called the \( s \)-decomposition of \( q \), such that \( \vec{w} \in \text{Var}(q_0) \) whenever \( k > 0 \), and

(i) \( q(\vec{x}) = q_0(\vec{x}, q_1(\vec{x}), \ldots, q_k(\vec{x})) \),
(ii) \( \nu(q, z) \leq s \) then \( z \notin \bigcup \text{Var}(q_{i+1}) \),
(iii) \( \mu \in \bigcup \text{Var}(q_{i+1}) \) then \( \nu(q, z) = \max\{\nu(q_1, z), \ldots, \nu(q_k, z)\} \),
(iv) \( \nu(q_{i+1}, z) \geq s + 1 \) for \( z \in \text{Var}(q_{i+1}) \),
(v) \( \nu(q_0) \leq s + 1 \).
Proof. We proceed by induction on the structure of q. If q is a constant or a variable, then s-dcp(q) := q, ∅ will do. So consider the case where q is q′ ◦ q″. The induction hypothesis provides suitable s-decompositions q′(¯x, ¯u, v), q′′(¯x, v) and q′′(¯x, v), q′′(¯x, v), q′′(¯x, v), where ¯u ∩ v = ∅. If ◦ is max, or ◦ is × with r ≤ s, or if ◦ is + with r ≤ s + 1, then we define

\[ s-dcp(q) := q_0(\bar{x}, \bar{u}, \bar{v}), q'_1(\bar{x}), \ldots, q'_k(\bar{x}), q''_1(\bar{x}), \ldots, q''_l(\bar{x}) \]

where q_0(\bar{x}, \bar{u}, \bar{v}) := q'_0(\bar{x}, \bar{u}) ◦ q''_0(\bar{x}, \bar{v}). Now (i), . . . , (iv) follow easily from the induction hypothesis. Otherwise if ◦ is × with r > s, or if ◦ is + with r > s + 1, then we define s-dcp(q) := w, q for some new variable w. Here (i), . . . , (v) are obvious.

Lemma 3.4.11 (Separation). For every Grzegorczyk polynomial q(\bar{x}) and integer r ≥ 0 one can find a Grzegorczyk polynomial ¯q over \bar{x} := \{ x | \nu(q, x) > r \} such that

1. \[ q(\bar{x}) ≤ \bar{q}(\bar{x}) + \nu(\bar{x}) \]
2. \[ \nu(q, x) = \nu(q, x) \] for all x ∈ \bar{x}.

Proof. Induction on \nu(q). In the base case \nu(q) = 0 we obtain q(\bar{x}) ≤ c_q + \nu(\bar{x}) by Domination (3.4.6) and Lemma 3.4.2 (d). In that case \bar{q} := q_0 and \bar{x} := \nu(q) will do.

Step case \nu(q) > 0. If r ≥ \nu(q) then q(\bar{x}) ≤ c_q + \nu(\bar{x}) by Domination (3.4.6) and Lemma 3.4.2 (d), hence \bar{q} := q_0 and \bar{x} := \nu(q) will do. If r = \nu(q) − 1 then we apply Top Level Separation (3.4.8) to obtain a Grzegorczyk polynomial q′(\bar{x}) such that q(\bar{x}) ≤ q′(\bar{x}) + \nu(q′) \max \bar{x} for some \bar{x} ⊆ \nu(q) \bar{x}, and \nu(q′, x) = \nu(q) + 1 for x ∈ \bar{x}. Hence \bar{q} := q′ will do.

So assume that r < \nu(q) − 1. In that case we apply Decomposition (3.4.10) to obtain the r-decomposition r-dcp(q) = q_0(\bar{x}w), q_1(\bar{x}), . . . , q_k(\bar{x}) of q, satisfying (i), . . . , (v). As \nu(q_0) ≤ r + 1 < \nu(q), the I.H. on q_0 yields a Grzegorczyk polynomial \bar{q}_0 over \bar{x}w = \{ u | \nu(q_0, u) = r + 1 \} satisfying

1. q_0(\bar{x}w) ≤ q_0(\bar{x}w) + \nu(\bar{x}w) for some \bar{x}w ⊆ \nu(q_0) \bar{x}w,
2. \nu(q_0, u) = \nu(q_0, u) = r + 1 for all u ∈ \bar{x}w.

Let \bar{x}_j be \nu(q_j) for j = 1, . . . , k. Then we define \bar{q} by

\[ \bar{q} := q_0(\bar{x}_1, q_1(\bar{x}_2), . . . , q_k(\bar{x}_k)) + \nu(\bar{x}_1, . . . , q_k(\bar{x}_k)) \]

where it is understood that q_j+1(\bar{x}_{j+1}) is only substituted for w_{j+1} in q_0(\bar{x}w) if w_{j+1} is in \bar{x}w. Furthermore, let \bar{x} be the x_i’s in \bar{x}w. Then we obtain:

\[ q(\bar{x}) = q_0(\bar{x}, q_1(\bar{x}_1), . . . , q_k(\bar{x}_k)) \]
\[ ≤ q_0(\bar{x}, q_1(\bar{x}_1), . . . , q_k(\bar{x}_k)) + \nu(\bar{x}_1, . . . , q_k(\bar{x}_k)) \]
\[ = \bar{q} + \nu(\bar{x}) \]

by (i)

Thus, to finish this case, it remains to show \bar{x} ⊆ \nu(q) and part (b). First consider any x ∈ \bar{x}, that is, \nu(q, x) ≥ r + 1. If x ∈ \bigcup \nu(q_j+1) then \nu(q, x) = \max(\nu(q_1, x), . . . , \nu(q_k, x)) by (iii), and that value is \nu(\bar{q}, x); because by (2) every u ∈ \nu(q_0) has modified measure r + 1, and by (iv) every u ∈ \nu(q_j+1) has a modified measure ≥ r + 1. Otherwise if x /∈ \bigcup \nu(q_j+1) then r + 1 ≤ \nu(q, x) by Composition (3.4.9) and (i). Thus we conclude from (v), (2) that r + 1 = \nu(q_0, x) = \nu(q_0, x) = \nu(\bar{q}, x). In particular, this shows \bar{x} ⊆ \nu(q); the converse holds by a similar argument.
Lemma 3.4.12 (Modified Bounding). For every term \( t \in \text{PR}_1 \) and every list \( \vec{x} := x_1, \ldots, x_p \) of distinct variables such that \( \vec{x} \supseteq \text{FV}(t) \) and \( \mu(\lambda \vec{x}.t) = (\tau_1, \ldots, \tau_m \rightarrow \iota) \), one can find a Grzegorczyk polynomial \( q_t(u_1, \ldots, u_m) \) satisfying

\[
\begin{align*}
(\alpha) & \quad [\lambda \vec{x}.t](\vec{a}, \vec{b}) \leq q_t(\vec{a}, \vec{b}) \quad \text{for } t \in \text{PR}_1 \\
(\beta) & \quad [\lambda \vec{x}.t](\vec{a}, \vec{b}) \leq q_t(\vec{a}, \vec{b}) \quad \text{for } t \in \text{PR}_2 \\
(\gamma) & \quad \nu(q_t, u_i) = R(\iota_i) \quad \text{for } 1 \leq i \leq m_t \\
(\delta) & \quad u_i \in \text{Var}(q_t) \iff \iota \neq \iota_i \quad \text{for } 1 \leq i \leq m_t.
\end{align*}
\]

Proof. We write \( \mu(\lambda \vec{x}.t) = (\tau_1, \ldots, \tau_m \rightarrow \iota) \) to explicitly indicate the reference to the term \( t \) under consideration. Furthermore, it is understood that the numbering \( u_1, \ldots, u_m \) of \( q_t \)'s input variables corresponds to the input positions of \( \lambda \vec{x}.t \), that is, \( u_1, \ldots, u_p \) refer to \( \vec{x} := x_1, \ldots, x_p \) and hence to \( \tau_1, \ldots, \tau_p \), and \( u_{p+1}, \ldots, u_m \) to the (already existing) “input positions” of \( t \), that is, to \( \tau_{p+1}, \ldots, \tau_m \).

We prove the Modified Bounding lemma for \( \text{PR}_1 \), and discuss the minor changes for \( \text{PR}_2 \) afterwards – indeed the proof for \( \text{PR}_2 \) is practically the same. We proceed by induction on the structure of \( t \in \text{PR}_1 \), in each case assuming an arbitrary but fixed list \( \vec{x} := x_1, \ldots, x_p \) of distinct variables such that \( \vec{x} \supseteq \text{FV}(t) \).

If \( t \) is a variable \( x_i \), then we set \( q_t := u_i \). If \( t \) is the constant 0 then \( q_t := 0 \) will do, while if \( t \) is the constant \( \text{S} \) then we set \( q_t := 1 \). If \( t = \lambda \vec{x}.r \) then we apply the induction hypothesis on \( r \) and \( \vec{x}, x \) to obtain a suitable Grzegorczyk polynomial \( q_r \). Hence \( q_t := q_r \) will do.

Case \( t = rs \) where \( \mu(r) = X; (\tau_1, \ldots, \tau_{m_r} \rightarrow \iota) \) and \( \mu(s) = Y; \iota \). The induction hypothesis provides suitable Grzegorczyk polynomials \( q_r(u_1, \ldots, u_{m_r}) \) and \( q_s(u_1, \ldots, u_p) \), both with respect to \( \vec{x} \). Hence by monotonicity of Grzegorczyk polynomials, we obtain:

\[
[\lambda \vec{x}.r](\vec{a}, \vec{b}) = [\lambda \vec{x}.r][\lambda \vec{x}.s](\vec{a}, \vec{b}) \leq q_r(\vec{a}, q_s(\vec{a}), \vec{b})
\]

If \( \tau_{p+1} = \iota \) then \( p + 1 \) is a redundant input position of \( \lambda \vec{x}.r \) by Redundancy (3.3.4). Thus, in this case we can define \( q_t := q_r(u_1, \ldots, u_p, u_{p+2}, \ldots, u_{m_r}) \), as \( \mu(\lambda \vec{x}.t) = (\tau_1, \ldots, \tau_p, \tau_{p+2}, \ldots, \tau_{m_r} \rightarrow \iota) \). So consider the case where \( \tau_{p+1} = \iota \). Hence \( \mu(rs) = X \cup Y; (\tau_1, \ldots, \tau_{m_r} \rightarrow \iota) \). Accordingly, we define \( q_t \) by

\[
q_t := q_r(u_1, \ldots, u_p, q_s(u_1, \ldots, u_p), u_{p+2}, \ldots, u_{m_r}).
\]

By (1) it remains to prove (b), (c). As to (b), for \( p + 1 < i \leq m_r \) we conclude from Composition (3.4.9) and the induction hypothesis on \( r \) that \( \nu(q_t, u_i) = \nu(q_r, u_i) = R(\iota_i) = R(\iota^*_i) \). As for \( 1 \leq i \leq p \), first note that

\[
R(\iota^*_i) = \max \{ \text{RANK}(x_i, X), \text{RANK}(x_i, Y) \}
\]

Since \( \tau_{p+1} = \iota \), the induction hypothesis on \( r \) yields \( u_{p+1} \in \text{Var}(q_r) \) and \( \nu(q_r, u_{p+1}) = \iota \). Thus we conclude from Composition (3.4.9) that

\[
\nu(q_t, u_i) = \max \{ \nu(q_r, u_i) \cup \{ l, \nu(q_s, u_i) \mid u_i \in \text{Var}(q_s) \} \}
\]

The induction hypothesis on \( r, s \) gives \( u_i \in \text{Var}(q_s) \iff \iota^*_i \neq \iota \iff x_i \in \text{VAR}(Y) \), and \( \nu(q_r, u_i) = R(\iota^*_i) = \text{RANK}(x_i, X) \) and \( \nu(q_s, u_i) = R(\iota^*_i) = \text{RANK}(x_i, Y) \). Thus, \( \nu(q_t, u_i) = R(\iota_i) \).
As to (c), for \( p + 1 < i \leq m_s \) the induction hypothesis on \( r \) yields \( u_i \in \text{Var}(q_r) \Leftrightarrow u_i \in \text{Var}(q_r) \Leftrightarrow i \neq \tau^r_i \). For \( 1 \leq i \leq p \) the induction hypothesis on \( r, s \) together with \( u_{p+1} \in \text{Var}(q_r) \) and \( \tau^r_{p+1} \neq i \) gives \( u_i \in \text{Var}(q_r) \Leftrightarrow u_i \in \text{Var}(q_r) \cup \text{Var}(q_s) \Leftrightarrow \tau^r_i \neq i \lor \tau^s_i \neq i \Leftrightarrow \tau^r_i \neq i \). This concludes the application case.

Case \( t \) is a recursion \( r s \). Hence \( f := [\lambda \vec{x}. t] \) satisfies \( f(\vec{a}, 0) = g(\vec{a}) \) and \( f(\vec{a}, b + 1) = h(\vec{a}, b, f(\vec{a}, b)) \) where \( g := [\lambda \vec{x}. r] \) and \( h := [\lambda \vec{x}. s] \). Suppose that \( \mu(r) = X \setminus \iota \) and \( \mu(s) = Y : (\tau^s_{p+1}, \tau^m_{m_s} \to \iota) \). The I.H. provides suitable Grzegorczyk polynomials \( q_r(u_1, \ldots, u_p) \) and \( q_s(u_1, \ldots, u_p, u_{p+1}, u_{m_s}) \).

Subcase \( \tau^m_{m_s} = \iota \). Hence \( \mu(t) = X \cup Y ; (r(\tau^s_{p+1}) \to \iota) \) and \( u_{m_s} \not\in \text{Var}(q_s) \) by the induction hypothesis on \( s \), and \( m_s \) is a redundant input position of \( h \) by Redundancy (3.3.4). The latter implies

\[
[\lambda \vec{x}. t](\vec{a}, b) \leq \max[q_r(\vec{a}), q_s(\vec{a}, b)]
\]

and so we can define \( q_t(u_1, \ldots, u_{p+1}) := \max(q_r(u_1, \ldots, u_p), q_s(u_1, \ldots, u_p, u_{p+1})) \). Using the induction hypothesis on \( r, s \), one easily verifies that (a),(b),(c) are true of \( q_t \).

Subcase \( \tau^m_{m_s} = l \). Hence \( \nu(q_s, u_{m_s}) = l \) by the induction hypothesis on \( s \), and furthermore,

\[
\mu(t) = X \cup Y_l; (k \to l)
\]

where \( k = \max(R(\tau^s_{p+1}), l + 1) \). Now we apply Separation (3.4.11) to \( q_s \) and \( l \) to obtain a Grzegorczyk polynomial \( \tilde{q}_s \) over \( \vec{u}^* := \{ u \mid \nu(q_s, u) \geq l + 1 \} \) such that

\[
q_s(\vec{u}) \leq \tilde{q}_s(\vec{u}^*)^{l+1} \max(\vec{u}^*, u_{m_s})
\]

\[
\nu(\tilde{q}, u) = \nu(q_s, u) \text{ for every } u \in \vec{u}^*
\]

where \( \vec{u}^* := \{ u \mid \nu(q_s, u) \leq l \} \) \( \setminus \{ u_{m_s} \} \). Then we define \( q_t \) by

\[
q_t(u_1, \ldots, u_p, u_{p+1}) := (u_{p+1} \times 1 \tilde{q}_s(\vec{u}^*^{l+1}) + \max(\vec{u}^*, q_r(u_1, \ldots, u_p))
\]

Recall that \( \times 1 \) is the ordinary multiplication. To prove \( f(\vec{a}, b) \leq q_t(\vec{a}, b) \), we proceed by induction on \( b \). In the base case we obtain \( f(\vec{a}, 0) \leq q_t(\vec{a}, 0) \) by the induction hypothesis on \( r \) and Lemma 3.4.2. In the step case \( b \to b + 1 \), we conclude (a) as follows:

\[
f(\vec{a}, b+1) \leq q_t(\vec{a}, b, f(\vec{a}, b)) \leq \tilde{q}_s(\vec{a}b) + 1^{l+1} \max(\vec{a}b^*, q_t(\vec{a}, b)) = S^{l+1}q_s(\vec{a}b) \leq \max(\vec{a}b^*, q_r(\vec{a}, b)) \]

As for (b), if \( u_{p+1} \in \vec{u}^* \) then \( \nu(q_s, u_{p+1}) = \nu(q_s, u_{p+1}) \geq l + 1 \) by (4). Thus, (2) and the induction hypothesis on \( qs \) imply \( R(\tau^m_{m_s}) = k = R(\tau^s_{p+1}) = \nu(q_s, u_{p+1}) = \nu(q_t, u_{p+1}) \). Otherwise if \( u_{p+1} \not\in \vec{u}^* \), then (2) and the induction hypothesis on \( qs \) imply \( R(\tau^s_{m_s}) = k = l + 1 = \nu(q_t, u_{p+1}) \).

As for a parameter \( u_i \) (\( 1 \leq i \leq p \)), first consider the case \( u_i \in \vec{u}^* \). Hence \( \nu(\tilde{q}_s, u_i) = \nu(q_s, u_i) \geq l + 1 \) by (4). Therefore the induction hypothesis implies \( \nu(q_t, u_i) = \max(\nu(q_s, u_i), \nu(q_t, u_i)) \geq l + 1 \) by the induction hypothesis on \( s \). If \( u_i \in \vec{u}^* \), then the I.H. on \( q_s \) gives \( \nu(q_s, u_i) = \max(l, \nu(q_r, u_i)) = \max(l, R(\tau^r_i)) \), and this value is also \( R(\tau_i^r) \). Otherwise if \( u_i \not\in \vec{u}^* \), then \( u_i \not\in \text{Var}(q_s) \), hence \( \tau^r_i = \iota \) by the induction hypothesis on
s. In that case the definition of modified measure \( \nu \) and the induction hypothesis on \( q_r \) imply \( \nu(q_t, u_i) = \max\{l, \nu(q_t, u_i) \mid u_i \in \text{Var}(q_t)\} = \max\{l, R(\tau^+_t) \mid \tau^+_t \neq \tau \}. \) This value is also \( R(\tau^+_t) \), concluding the proof of (b).

Concerning (c), first note that \( u_{p+1} \in \text{Var}(q_t) \) and \( \tau^+_t = k \). As for a parameter \( u_i \), first observe that \( u_i \in \text{Var}(q_t) \Leftrightarrow u_i \in \text{Var}(q_s) \lor u_i \in \vec{u}^s \). Thus we conclude from the induction hypothesis on \( r, s \) that \( u_i \in \text{Var}(q_t) \Leftrightarrow u_i \in \text{Var}(q_s) \Leftrightarrow R(\tau^+_t) \neq \tau \lor R(\tau^+_t) \neq \tau \Leftrightarrow R(\tau^+_t) \neq \tau \). This completes the proof of the Modified Bounding lemma for PR_{1.}

The proof of the lemma for PR_{2} differs slightly in the case where \( t = \text{RN}_r t_0 t_1 \). In that case \( \lambda \vec{x}.t \) defines the function \( f \) satisfying \( f(\vec{a}, 0) = g(\vec{a}) \) and \( f(\vec{a}, s_i(b)) = h_i(\vec{a}, b, f(\vec{a}, b)) \) for \( s_i(b) \neq 0 \), where \( g := [\lambda \vec{x}.r] \) and \( h_i := [\lambda \vec{x}.t_i] \). The induction hypothesis yields Grzegorczyk polynomials \( q_r(u_1, \ldots, u_p) \) and \( q_h(u_1, \ldots, u_p, u_{p+1}, u_{p+2}) \). Now let \( q := \max(q_{t_0}, q_{t_1}) \). Then one carries out all arguments for \( q \) instead of \( q_s \) in the previous recursion case. Due to the point-wise definition of \( \nu \) for Grzegorczyk polynomials \( \max(q_1, q_2) \), all arguments for \( q_s \) carry over to \( q \). More precisely, if \( \tau^0_{m+p+2} = \tau \) and \( \tau^1_{m+p+2} = \tau \), we define

\[
q_t(u_1, \ldots, u_p, u_{p+1}) := \max(q_r(u_1, \ldots, u_p), q(u_1, \ldots, u_p, u_{p+1})).
\]

Otherwise if \( \tau^0_{m+p+2} \neq \tau \) or \( \tau^1_{m+p+2} \neq \tau \), Separation (3.4.11) for \( q := \max\{R(\tau^0_{m+p+2}), R(\tau^1_{m+p+2})\} \) yields a Grzegorczyk polynomial \( q \) satisfying (3),(4) above. Now let

\[
q_t(u_1, \ldots, u_p, u_{p+1}) := (u_{p+1} \times 1 q(\vec{a})) + \max_{\{l \mid \text{Var}(\vec{x}) \}}(\vec{a}) + 1, l+1 \max_{\{l \mid \text{Var}(\vec{x}) \}}(\vec{a}), q_r(u_1, \ldots, u_p)).
\]

To prove \( |f(\vec{a}, b)| \leq q_t(|\vec{a}, b|) \), one uses induction on \( |b| \), employing the same steps as above. \( \square \)

**Theorem 3.4.13 (Bounding).** For every closed term \( t \in \text{PR}_i \) of type \( \vec{x} \rightarrow \tau \) one can find a Grzegorczyk polynomial \( q_t \in \mathcal{E}_{\mu(t)+1} \) satisfying \( \|t\|(<\vec{x}) = q_t(<\vec{x}) \) for \( t \in \text{PR}_1 \), and \( \|t\|(<\vec{x}) \leq q_t(<\vec{x}) \) for \( t \in \text{PR}_2 \).

**Proof.** Suppose that \( \mu(t) = (\tau_1, \ldots, \tau_m \rightarrow \tau) \), and let \( q_t(u_1, \ldots, u_m) \) be the Grzegorczyk polynomial obtained from Modified Bounding (3.4.12). Then \( \nu(q) = \max\{\nu(q, u_i) \mid i = 1, \ldots, m\} = \max\{R(\tau_i) \mid i = 1, \ldots, m\} = \mu(t) \). Hence \( q_t \in \mathcal{E}_{\mu(t)+1} \) by Definability (3.4.5), and we are done. \( \square \)

### 3.5. Characterisation Theorems

The measure \( \mu \) on \( \text{PR}_i \) gives rise to the following hierarchy of modified Heinermann classes \( \mathcal{R}_i^n \). This section is then concerned with relating these classes to the Grzegorczyk hierarchy.

**Definition 3.5.1.** The \( n \text{th modified Heinermann class } \mathcal{R}_i^n \) is defined by

\[
\mathcal{R}_i^n := \{f \in \mathcal{PR} \mid f = \|t\| \text{ for some closed } t \in \text{PR}_i \text{ with } \mu(t) \leq n\}.
\]

In other words, a function belongs to “modified level” \( n \) if it can be defined by at most \( n \) top recursions, but with as many side or flat recursions as desired.

**Note 2.** The modified Heinermann classes \( \mathcal{R}_i^n \) are naturally closed under composition and side recursion.

First the results of Schwichtenberg and Müller are improved by showing \( \mathcal{E}_{n+1} = \mathcal{R}_1^n \) for \( n \geq 1 \). In contrast to the former, the proof of the inclusion \( \subseteq \) does not refer to any machine-based computation model. The “simulation method” presented here allows one to transform directly every definition of a function \( f \in \mathcal{E}_{n+1} \) into one in \( \mathcal{R}_1^n \). The method consists in separating the “structure” in the definition of \( f \) from the
growth rate of \( f \), that is, we show that each such \( f \) has a “simulation” \( f^* \) in \( R_1^1 \) and a “witness” \( w_f \) in \( R_1^n \) such that \( f^*(w, \vec{x}) = f(\vec{x}) \) whenever \( w \geq w_f(\vec{x}) \). Thus, by composing \( f^* \) with \( w_f \) one obtains a definition in \( R_1^n \). The method generalises Bellantoni’s proof [3] that the functions in \( E_2 \) can be simulated by safe primitive recursion. Since \( R_1^1 \) is closed under composition, this improves the result in [3]. The inclusion \( \subseteq \) follows from the Bounding Theorem by a straightforward inductive argument, because Bounding (3.4.13) turns every primitive recursion in \( R_1^n \) into a bounded primitive recursion in \( E_{n+1} \).

In the sequel we sometimes mix closed terms \( f \in PR_1 \) with the functions \([f]\) denoted.

**Lemma 3.5.2 (E-Bounding).** Every \( f \in E_{n+2} \) has a monotone increasing bound \( b_f \in R_1^{n+1} \).

**Proof.** First recall that \( A_2 \equiv x^1 \) and \( \mu(x^1) = (1, 1 \rightarrow i) \). Under this definition of \( A_2 \), \( A_{m+3} \) is defined inductively by a top recursion from \( A_{m+2} \), that is, \( A_{m+3} := \lambda xy.R \ S_0 (\lambda uv.A_{m+2} x y) y \). Thus, we obtain

\[
\mu(A_{m+3}) = (m + 1, m + 2 \rightarrow i).
\]

Note that \( \max \) is in \( R_1^1 \), as \( \max = \lambda xy.C (\lambda xy)x y \) where \( \lambda xy.R x (\lambda uv.P v) y \) defines modified subtraction \( - \), and \( C, P \) are as in note 1. Recall that every \( f \in E_{n+2} \) satisfies \( f(\vec{x}) \leq A_{n+3}(\max(2, \vec{x}), c_f) \) for some constant \( c_f \). Thus, by unfolding the constant-depth recursion, that is, \( A_{n+3}(\max(2, \vec{x}), c_f) = A_{n+2}(\max(2, \vec{x}), \ldots, A_{n+2}(\max(2, \vec{x}), 1) \ldots) (c_f - 1 \text{ calls of } A_{n+2}) \), we obtain a monotone increasing bound on \( f \) in \( R_1^{n+1} \). Observe that the unfolding does stop at this level, since \( A_{n+2}(x, 1) = x \).

**Theorem 3.5.3 (At and above Linear Space).** \( \forall n \geq 1. E_{n+1} = R_1^n \)

**Proof.** The inclusion \( \subseteq \) follows from (A) below, for given a function \( f \in E_{n+1} \), \( n \geq 1 \), then (A) implies \( f(\vec{x}) = f^*(w_f(\vec{x}), \vec{x}) \) for some \( f^* \in R_1^1 \) and \( w_f \in R_1^n \).

(A) For every \( f \in E_{n+2} \) one can find a simulation \( f^* \in PR_1 \) and a monotone increasing witness \( w_f \in R_1^{n+1} \) such that \( f^*(w, \vec{x}) = f(\vec{x}) \) for \( w \geq w_f(\vec{x}) \), and \( \mu(f^*) = (\tau_0, \tau_1, \ldots, \tau_{m_f} \rightarrow i) \) with \( \tau(\tau_0) \leq 1 \), and \( R(\tau_i) = 0 \) for \( i = 1, \ldots, m_f \).

The proof of (A) is by induction on the definition of \( f \in E_{n+2} \). If \( f \) is one of the initial functions \( 0, S, \Pi_j^m \) then \( f^* := \lambda w.0, \lambda wx.S x, \lambda wy.x \) respectively and \( w_f := \lambda \vec{x}.0 \) will do. The case where \( f \) is the principal function \( A_{\ell+2} \) can be reduced to that of bounded primitive recursion, because \( A_{\ell+2} \) can be defined by bounded primitive recursion in \( E_{\ell+2} \) using \( A_{\ell+2} \) itself as bounding function.

Case \( f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_m(\vec{x})) \). The induction hypothesis yields simulations \( h^*, g_1^*, \ldots, g_m^* \) in \( PR_1 \) with monotone increasing witness functions \( w_h, w_{g_1}, \ldots, w_{g_m} \) in \( R_1^{\ell+1} \). By E-Bounding (3.5.2) we obtain a monotone increasing bound \( b \in R_1^{\ell+1} \) on all \( g_1, \ldots, g_m \). We define \( f^* \) and \( w_f \) by:

\[
\begin{align*}
    f^* & := \lambda w(\vec{x}).h^*(w(g_1^* w(\vec{x})) \ldots (g_m^* w(\vec{x}))) \\
    w_f & := \lambda \vec{x}.(w_h(\vec{x}) \ldots (b(\vec{x}))) + (w_{g_1}(\vec{x}) + \ldots + (w_{g_m}(\vec{x})))
\end{align*}
\]

To see that \( f^* \) defines a simulation of \( f \) with witness \( w_f \), observe that the assumption \( w \geq w_f(\vec{x}) \) implies \( w \geq w_h(g_1(\vec{x}), \ldots, g_m(\vec{x})) + w_{g_1}(\vec{x}) + \ldots + w_{g_m}(\vec{x}) \). Thus \( f^*(w, \vec{x}) = f(\vec{x}) \) follows from the induction hypothesis on \( g_1, \ldots, g_m, h \).

Case \( f(\vec{x}, 0) = g(\vec{x}) \) and \( f(\vec{x}, y + 1) = h(\vec{x}, y, f(\vec{x}, y)) \), and \( f \leq k \) where \( g, h, k \in E_{\ell+2} \). The induction hypothesis yields simulations \( h^*, g^* \) in \( PR_1 \) with monotone increasing witness functions \( w_h, w_g \) in \( R_1^{\ell+1} \). The required witness \( w_f \) can be defined by

\[
w_f := \lambda \vec{x}y. w_h(\vec{x} y(b(\vec{x} y)) + (w_g(\vec{x} y) + y
\]

\[
\begin{align*}
    f^* & := \lambda w(\vec{x}).h^*(w(g_1^* w(\vec{x})) \ldots (g_m^* w(\vec{x}))) \\
    w_f & := \lambda \vec{x}.(w_h(\vec{x}) \ldots (b(\vec{x}))) + (w_{g_1}(\vec{x}) + \ldots + (w_{g_m}(\vec{x})))
\end{align*}
\]
where \( b \in \mathcal{R}^{f+1} \) is a monotone increasing bound on \( k \) obtained from \( \mathcal{E} \)-Bounding (3.5.2). To obtain a suitable simulation \( f^* \) of \( f \), we first define a function \( \hat{f} \) in \( \mathcal{R}^1 \) which simulates \( f \) up to \( y \), that is,

\[
\begin{align*}
(i) & \quad \text{If } w \geq w_f(x, y) \text{ then } \hat{f}(\hat{w}, w, x, y) = \begin{cases} f(\hat{x}, \hat{w}) & \text{if } \hat{w} \leq y \\
 f(\hat{x}, y) & \text{else} \end{cases} \\
(ii) & \quad \mu(\hat{f}) = (\tau_0, \tau_1, \ldots, \tau_{m_f} \rightarrow \iota) \text{ such that } \mathcal{R}(\tau_0) \leq 1, \text{ and } \mathcal{R}(\tau_i) = 0 \text{ for } i = 1, \ldots, m_f.
\end{align*}
\]

Obviously, once \((i),(ii)\) are established, \( f^* := \lambda x y \bar{y}. \hat{f}_w \bar{x} y \) defines the required simulation of \( f \). According to \((i)\), we define the function \( \hat{f} \) by side recursion from \( g^*, h^* \) as follows:

\[
\begin{align*}
\hat{f}(0, w, x, y) & = g^*(w, x) \\
\hat{f}(\hat{w} + 1, w, x, y) & = \begin{cases} h^*(w, x, \hat{w}, \hat{f}(\hat{w}, w, x, y)) & \text{if } \hat{w} + 1 \leq y \\
 f(\hat{x}, \hat{w} + 1) & \text{else} \end{cases}
\end{align*}
\]

To achieve \((ii)\), we need to define the characteristic function of \( \mathcal{E} \), such that \( \mu(c^\leq) = (1, 0 \rightarrow \iota) \). Since \( x \leq y \Leftrightarrow y + 1 - x > 0 \), we can do this by setting \( c^\leq := \lambda xy. C(\bar{w}(S y)x) (S) \) (\( S \)), where \( C \) is defined as in note 1, and \( \mu(\bar{w}) = (0, 1 \rightarrow \iota) \). Thus, following the recursion equations of \( \hat{f} \), we define:

\[
\hat{f} := \lambda w x \bar{y} y. \mathcal{R}(g^* w \bar{x} y)(\lambda u v. C(c^\leq(S u)v)(h^* w \bar{x} u v)) \hat{w}
\]

One easily reads off this definition that it satisfies \((ii)\), concluding the proof of \( \leq \).

The remaining inclusion \( \geq \) follows from \( (B) \) below, because \( (B) \) implies \( \mathcal{R}_1^n \subseteq \mathcal{E}_{n+1} \) for all \( n \).

(B) For every \( t \in \mathcal{PR}_1 \) and every \( x \supseteq \mathcal{F}(t) \), \( [\lambda \bar{x}. t] \) belongs to \( \mathcal{E}_{\mu(t)+1} \).

By Fairness (3.3.10) we can prove \( (B) \) by induction on the structure of fair terms \( t \). If \( t \) is a variable or a constant \( 0, S \), we have \( [\lambda \bar{x}. t] \in \mathcal{E}_0 \). The case where \( t = \lambda x. r \) follows from the induction hypothesis on \( r \) with respect to \( \bar{x}, x \). The case where \( t \) is \( rs \) follows from the induction hypothesis on \( r \) and \( s \), because \([\lambda \bar{x}. t](\bar{a}, \bar{b}) = [\lambda \bar{x}. r](\bar{a}, [\lambda \bar{x}. s](\bar{a}), \bar{b}) \) and fairness of \( t \) implies \( \mu(t) = \max(\mu(r), \mu(s)) \). So consider the case where \( t \) is \( \mathcal{R} r s \). The induction hypothesis on \( r \) and \( s \) implies \([\lambda \bar{x}. r], [\lambda \bar{x}. s] \in \mathcal{E}_{\mu(t)+1} \), and Bounding (3.4.13) gives that \([\lambda \bar{x}. t] \) has a bound in \( \mathcal{E}_{\mu(t)+1} \). Thus, \([\lambda \bar{x}. t] \) is defined in \( \mathcal{E}_{\mu(t)+1} \).

A further benefit of the Bounding Theorem, together with the ability of \( \mu \) to recognise flat recursions, is a simple proof that \( \mathcal{R}_2^1 \) characterises the polynomial-time computable functions. Since \( \mathcal{R}_2^1 \) is closed under composition, this streamlines and improves both the result and proof in [5]. The proof is based on Cobham’s [16] characterisation of \( \mathcal{F} \) (cf. Background 2.5).
Lemma 3.5.4 ($\mathcal{R}_2^1$ Closure). For ordinary polynomials $q(\vec{x})$ the function $\lambda \vec{x}.2^{q(|\vec{x}|)}$ is in $\mathcal{R}_2^1$.

Proof. Rereading the proof of FP-Closure (cf. Background 2.5), stating that $\lambda \vec{x}.2^{q(|\vec{x}|)}$ belongs to FP for ordinary polynomials $q(\vec{x})$, it suffices to show that smash can be defined in $\mathcal{R}_2^1$. To see this, first consider the function shift-left satisfying $\text{shift-left}(x, y) = s_0^{(|x|)}(y) = 2^{|x|} \cdot y$ can be defined by

$$\text{shift-left} := \lambda x. y. \text{RN} y (\lambda uv.s_0 v)(\lambda uv.s_0 v) \; x$$

such that $\mu(\text{shift-left}) = (1, 0 \rightarrow i)$. Now, as $2^{|0|\cdot |y|} = 1$ and $2^{|s_i(x)|\cdot |y|} = 2^{|y|} \cdot 2^{|x|\cdot |y|} = \text{shift-left}(y, 2^{|x|\cdot |y|})$ for $s_i(x) \neq 0$, we define smash by side recursion from shift-left as follows:

$$\text{smash} := \lambda x. y. \text{RN} s_1 0 (\lambda uv.\text{shift-left} yv)(\lambda uv.\text{shift-left} yv) \; x$$

Accordingly, we obtain $\mu(\text{smash}) = (1, 1 \rightarrow i)$ and hence smash $\in \mathcal{R}_2^1$ as required. \hfill \square

Theorem 3.5.5 (Polynomial Time). $\mathcal{R}_1^1 = \text{FPTIME}$

Proof. First we prove $\subseteq$. Given $f \in \mathcal{R}_2^1$, Bounding (3.4.13) yields a Grzegorczyk polynomial $q_f$ satisfying $|f(\vec{x})| \leq q_f(|\vec{x}|)$ and $\nu(q_f) \leq 1$. The latter implies that $q_f$ is an ordinary polynomial. We conclude from $\mathcal{R}_2^1$ Closure (3.5.4) that $f$ can be bounded in FP by $\lambda \vec{x}.2^{q_f(|\vec{x}|)}$. Thus, every recursion on notation in $\mathcal{R}_2^1$ can be turned into a bounded recursion on notation in FP, implying $\subseteq$ by Cobham’s theorem. In fact, similar to the proof of (B), one obtains:

(C) For every $t \in \mathcal{R}_2$ with $\mu(t) \leq 1$, and $\vec{x} \supseteq \text{FV}(t)$, $[\lambda \vec{x}.t]$ belongs to FP.

The inclusion $\supseteq$ follows from (D) below, since each $f \in \text{FP}$ has a representation $f(\vec{x}) = f^*(w, \vec{x})$ whenever $|w| \geq p_f(|\vec{x}|)$, for some $f^* \in \mathcal{R}_2^1$ and some ordinary polynomial $p_f$. Thus, we conclude from $\mathcal{R}_2^1$ Closure (3.5.4) that $\lambda \vec{x}.f^*(2^{q_f(|\vec{x}|)})\vec{x}$ defines $f$ in $\mathcal{R}_2^1$.

(D) For every $f \in \text{FP}$ one can find a simulation $f^* \in \mathcal{R}_2$ and an ordinary polynomial $p_f$ such that $f^*(w, \vec{x}) = f(\vec{x})$ whenever $|w| \geq p_f(|\vec{x}|)$, and $\mu(f^*) = (\tau_0, \tau_1, \ldots, \tau_{m_f} \rightarrow i)$ with $\nu(\tau_0) \leq 1$, and $\text{R}(\tau_i) = 0$ for $i = 1, \ldots, m_f$.

The proof is by induction on the definition of $f \in \text{FP}$. In fact, the proof is just a binary version of the proof of (A). If $f$ is one of the initial functions $0, \Pi_j^m, s_0, s_1$, then we define $f^* := \lambda \vec{x}.f^*(w, \vec{x})$ and $p_f := \lambda \vec{x}.0$.

The case where $f$ is the principal function smash can be reduced to that of bounded recursion on notation below, because smash can be defined from shift-left by bounded recursion on notation in FP, using smash itself as bounding function. Observe that shift-left is defined in FP by bounded recursion on notation, since $\text{shift-left}(x, y) = 2^{|x|} \cdot y \leq 2^{|x|\cdot |y|} \leq \text{smash}(s_1(x), s_1(y))$.

If $f$ is defined by composition from $h, g_1, \ldots, g_m \in \text{FP}$, then one uses the induction hypothesis to define

$$f^* := \lambda \vec{x}.h^* w(g_1^* w \vec{x}) \ldots (g_m^* w \vec{x})$$

$$p_f(\vec{x}) := p_h(q(\vec{x}), \ldots, q(\vec{x})) + \sum_j p_{g_j}(\vec{x}),$$

where $q$ is a polynomial length bound on all $g_1, \ldots, g_m$ obtained from FP-Bounding (cf. Background 2.5).
Finally, suppose that \( f(0, \vec{x}) = g(\vec{x}), f(s_i(y), \vec{x}) = h_i(y, \vec{x}, f(y, \vec{x})) \) for \( s_i(y) \neq 0 \), and \( f \leq k \), where \( g, h, k \in \text{FP} \). The induction hypothesis yields simulations \( g^\ast, h_i^\ast, h_i^\ast \in \text{PR}_2 \) and suitable ordinary polynomials \( p_{h_0}, p_{h_1}, p_g \). Similar to the recursion case in the proof of (A), we define

\[
p_f(y, \vec{x}) := (p_{h_0} + p_{h_1})(y, \vec{x}, q_f(y, \vec{x})) + p_g(\vec{x}) + s_1(y)
\]

where \( q_f \) is a polynomial length bound on \( f \) obtained from FP-Bounding (cf. Background 2.5). Furthermore, to obtain a simulation \( f^\ast \) of \( f \), we first define a function \( \hat{f} \) in \( R_1^2 \) such that \( \mu(\hat{f}) = (1, 1, \tau_1, \ldots, \tau_m) \) with \( \tau_1, \ldots, \tau_m \in \{1, 0\} \), and \( \hat{f} \) simulates \( f \) as follows: For \( |w| \geq p_f(|\vec{x}, y|) \),

\[(*) \quad \text{if } |w| - |y| \leq |\hat{w}| \leq |w|, \text{ then } \hat{f}(\hat{w}, w, \vec{x}, y) = f(p(|w| - |\hat{w}|)(y), \vec{x}).\]

Then \( f^\ast := \lambda w.\hat{f}wy\vec{x} \) defines a simulation of \( f \), as \( \hat{f}(w, w, \vec{x}) = f(y, \vec{x}) \) whenever \( |w| \geq p_f(|\vec{x}, y|) \).

First we define the function \( Y \) satisfying \( Y(\hat{w}, w, y) = p(|w| - |\hat{w}|)(y) \) and \( \mu(Y) = (1, 1, 0 \rightarrow \iota) \). That is obtained by \( Y := \lambda w.w.y \circ (\hat{\circ}w)y \), where \( \circ \) satisfying \( x \circ y = p(|x|)(y) \) and \( \mu(\circ) = (1, 0 \rightarrow \iota) \) is defined from \( p \) (cf. Note 1) by \( \circ := \lambda xy.\text{RN}y(\lambda vx.pv)(\lambda vx.pv)x \). Observe that \( |x \circ y| = |y| - |x| \).

According to \((*)\) and \( |w| \geq p_f(|y, \vec{x}|) > |y| \), \( \hat{f} \) has the following recursion equations:

\[
\hat{f}(0, w, y, \vec{x}) = 0
\]

\[
\hat{f}(s_i(\hat{w}), w, y, \vec{x}) = \begin{cases} 
g^\ast(w, \vec{x}) & |s_i(\hat{w})| \leq |w| - |y| \\
h_i^\ast(w, Y(\hat{w}, w, y), y, \vec{x}, f(\hat{w}, w, y, \vec{x})) & |s_i(\hat{w})| > |w| - |y| \text{ and } Y(s_i(\hat{w}), w, y) \text{ is even} \\
h_i^\ast(w, Y(\hat{w}, w, y), \vec{x}, f(\hat{w}, w, y, \vec{x})) & |s_i(\hat{w})| > |w| - |y| \text{ and } Y(s_i(\hat{w}), w, y) \text{ is odd}
\end{cases}
\]

To see \((*)\), we assume \( |w| \geq p_f(|\vec{x}, y|) \) and proceed by induction on \( |\hat{w}| \) satisfying \( |w| - |y| \leq |\hat{w}| \leq |w| \). If \( |\hat{w}| = |w| - |y| \), then \( |\hat{w}| > 0 \) and \( Y(\hat{w}, w, y) = 0 \), hence \( \hat{f}(\hat{w}, w, y, \vec{x}) = g^\ast(w, \vec{x}) = g(\vec{x}) = f(0, \vec{x}) \). So assume \( |w| - |y| < |s_i(\hat{w})| \leq |w| \), hence \( Y(s_i(\hat{w}), w, y) > 0 \), say \( Y(s_i(\hat{w}), w, y) \) is even. Since \( p(Y(s_i(\hat{w}), w, y)) = Y(\hat{w}, w, y) \), and \( |w| \geq p_{h_0}(|y, \vec{x}, q_f(|y, \vec{x}|)|) \geq p_{h_0}(|Y(\hat{w}, w, y), \vec{x}|, f(\hat{w}, w, y, \vec{x})) \), the I.H. on \( \hat{w} \) and that on \( h \) imply \( \hat{f}(s_i(\hat{w}), w, \vec{x}, y) = h_i^\ast(w, Y(\hat{w}, w, y), \vec{x}, f(\hat{w}, w, y, \vec{x})) = f(Y(s_i(\hat{w}), w, y), \vec{x}) = f(Y(s_1(\hat{w}), w, y), \vec{x}) \).

Now observe that under the hypothesis of \((*)\), \( |\hat{w}| \leq |w| - |y| \Leftrightarrow |y| \leq |w| - |\hat{w}| \Leftrightarrow Y(\hat{w}, w, y) = 0 \). Therefore the function \( f \) can be realised as \( f := \lambda w.w.y.\text{RN}0H_0H_1\hat{w} \), where \( H_i \) is defined from binary cases \( c \) (cf. Note 1), satisfying \( \mu(c) = (0, 0, 0 \rightarrow \iota) \), as follows:

\[
H_1 := \lambda uv.c(Y(s_{01}(uwy))g^\ast wx)(h_i^\ast w(Yuwyv))\]

\[\square\]

**Theorem 3.5.6** (At and above Kalmár-elementary). \( \forall n \geq 2. R_2^n = R_1^n \)

**Proof.** As to the inclusion \( \supseteq \), first recall (cf. Background 2.4) the definition of the unary function \( \circ \) used to synchronise the number of steps in a recursion on notation on \( \circ(x) \) with the number of steps in a primitive recursion on \( x \), that is, \( |\circ(x)| = x \). This function is defined in \( R_2^2 \) by a top recursion from concatenation \( \oplus \):

\[
\circ := \lambda x.\text{RN}0(\lambda yz.\oplus zz)(\lambda yz.s_1(\oplus zz))x.
\]

Recall that \( x \oplus y = x \cdot 2|y| + y \). This implies \( \circ(x) = 2^x - 1 \), and hence \( |\circ(x)| = x \) and \( p(\circ(x)) = \circ(x - 1) \). Now the inclusion \( \supseteq \) follows from (E) below, for given a closed term \( t \in \text{PR}_1 \), (E) yields a closed term \( t' \in \text{PR}_2 \) satisfying \( |t'| = |t| \circ(b) | = |t| \circ(b) \), and \( t, t' \) have identical \( \mu \)-measures. As \( | \cdot | \) belongs to \( R_1^2 \) by theorem 3.5.5, we conclude \( R_2^n \supseteq R_1^n \) for \( n \geq 2 \).
(E) For every $t \in \text{PR}_1$ and $\vec{x} \supseteq \text{FV}(t)$ with $\mu(\lambda \vec{x}.t) = (\tau_1, \ldots, \tau_m \rightarrow \iota)$ one can find a simulation $t^* \in \text{PR}_2$ such that $\llbracket \lambda \vec{x}.t^* \rrbracket(\circ(\vec{a}, \vec{b})) = \circ(\llbracket \lambda \vec{x}.t \rrbracket(\vec{a}, \vec{b}))$, and $\mu(\lambda \vec{x}.t^*) = (\tau_1, \ldots, \tau_m \rightarrow \iota)$.

The prove of (E) is by recursion on the structure of $t \in \text{PR}_1$, where in each case we will have $\text{FV}(t) = \text{FV}(t^*)$ (cf. Background 2.4. for correctness). If $t$ is $0, S, x_i$ then let $t^*$ be $0, s_1, x_i$ respectively. Otherwise:

$$(\lambda x.r)^* := \lambda x.r^*$$
$$(rs)^* := r^*s^*$$
$$(R r s)^* := RN r^*s^*s^*$$

The remaining inclusion $\subseteq$ follows from theorem 3.5.3 and (F) below.

(F) For every $t \in \text{PR}_2$ and $\vec{x} \supseteq \text{FV}(t)$, $\llbracket \lambda \vec{x}.t \rrbracket$ belongs to $\mathcal{E}_{\max(3,\mu(t)+1)}$.

By Fairness (3.3.10) we can prove (F) by induction on the structure of fair terms $t \in \text{PR}_2$. All cases are as in the proof of (B), except the case where $t$ is a recursion $RN r t_0 t_1$. By the induction hypothesis the functions $g := \llbracket \lambda \vec{x}.r \rrbracket, h_0 := \llbracket \lambda \vec{x}.t_0 \rrbracket, h_1 := \llbracket \lambda \vec{x}.t_1 \rrbracket$ all belong to $\mathcal{E}_{\max(3,\mu(t)+1)}$. So $\llbracket \lambda \vec{x}.t \rrbracket$ represents the function $f$ defined by recursion on notation from $g, h_0, h_1$ and auxiliary functions (all in $\mathcal{E}_1$) a function $\hat{f}$ satisfying

$$\hat{f}(\vec{x}, y, z) = f(\vec{x}, p^{(|y|-z)}(y)) \text{ for } z \leq |y|$$
$$\hat{f}(\vec{x}, y, z) = f(\vec{x}, y) \text{ for } z \geq |y|.$$

Hence $f(\vec{x}, y) = \hat{f}(\vec{x}, y, |y|)$, and so we obtain $f \in \mathcal{E}_{\max(3,\mu(t)+1)}$ provided $\hat{f}$ has a bound in $\mathcal{E}_{\max(3,\mu(t)+1)}$. To see this, Bounding (3.4.13) yields a Grzegorczyk polynomial $q \in \mathcal{E}_{\mu(t)+1}$ satisfying $|f(\vec{x}, y)| \leq q(|\vec{x}, y|)$.

Thus, by Monotonicity (3.4.2 (d)) and $|p^{(|y|-z)}(y)| \leq |y|$ we obtain

$$\hat{f}(\vec{x}, y, z) \leq 2^{q(|\vec{x}, y|)}.$$

This gives the required bound in $\mathcal{E}_{\max(3,\mu(t)+1)}$. \qed
4. The measure $\mu$ on stack and loop programs

In this chapter two imperative programming languages are considered, both having for-do type loops as the main control structure: One is a stack language which operates with stacks over a fixed, but arbitrary alphabet, the other is a slight modification of a loop language studied intensively in the literature, e.g. Meyer and Ritchie [40], Machthey [39] and Goetze and Nehrlich [22].

To each program $P$ in these languages, a measure $\mu(P)$ is assigned in a purely syntactic fashion. It is shown that stack programs of $\mu$-measure $n$ compute precisely the functions computable by a Turing machine in time bounded by a function in Grzegorczyk class $E^{n+2}$. Furthermore, loop programs of $\mu$-measure $n$ compute exactly the functions in $E^{n+2}$.

4.1. Stack programs

We presuppose a fixed, but arbitrary alphabet $\Sigma := \{a_1, \ldots, a_l\}$. We will define a stack programming language over $\Sigma$ where programs are built from primitive instructions $\text{push}(a,X)$, $\text{pop}(X)$, $\text{nil}(X)$ by sequencing, conditional statements and loop statements. We assume an infinite supply of variables $X$, $Y$, $Z$, $O$, $U$, $V$, possibly with subscripts. Intuitively, variables serve as stacks, each holding an arbitrary word over $\Sigma$ which can be manipulated by running a stack program.

Definition 4.1.1 (Stack programs). Stack programs $P$ are inductively defined as follows:

- Every imperative $\text{push}(a,X)$, $\text{pop}(X)$, $\text{nil}(X)$ is a stack program.
- If $P_1$, $P_2$ are stack programs, then so is the sequence statement $P_1; P_2$.
- If $P$ is a stack program, then so is every conditional statement $\text{if} \top(X) \equiv a [P]$.
- If $P$ is a stack program with no occurrence of $\text{push}(a,X)$, $\text{pop}(X)$ or $\text{nil}(X)$, then so is the loop statement $\text{foreach} X [P]$.

We use $V(P)$ to denote the set of variables occurring in $P$.

Note 4.1.2. Every stack program can be written uniquely in the form $P_1; \ldots; P_k$ such that each component $P_i$ is either a loop or an imperative, or else a conditional, and where $k = 1$ whenever $P$ is an imperative or a loop or a conditional.

We will use informal Hoare-like sentences to specify or reason about stack programs, that is, we will use the notation $\{A\} P \{B\}$, the meaning being that if the condition given by the sentence $A$ is fulfilled before $P$ is executed, then the condition given by the sentence $B$ is fulfilled after the execution of $P$. For example,

$$\{\vec{X} = \vec{w}\} P \{\vec{X} = \vec{w}'\}$$

reads: If the words $\vec{w}$ are stored in stacks $\vec{X}$, respectively, before the execution of $P$, then $\vec{w}'$ are stored in $\vec{X}$ after the execution of $P$. Another typical example is

$$\{\vec{X} = \vec{w}\} P \{|X_1| \leq f_1(|\vec{w}|), \ldots, |X_n| \leq f_n(|\vec{w}|)\}$$

meaning that if the words $\vec{w}$ are stored in stacks $\vec{X}$, respectively, before the execution of $P$, then each word stored in $X_i$ after the execution of $P$ has a length bounded by $f_i(|\vec{w}|)$. Here $f_i$ is any function over $\mathbb{N}$, and $|\vec{w}|$ abbreviates as usual the list $|w_1|, \ldots, |w_n|$.
Definition 4.1.3 (Operational semantics of stack programs). Imperatives, conditionals and loops in programs $P_1; \ldots; P_k$ are executed one by one from the left to the right, where

- the operational semantics of imperatives and conditionals is as expected:
  - $\text{push}(a, X)$ pushes letter $a$ on top of stack $X$,
  - $\text{pop}(X)$ removes the top symbol on stack $X$, if any, otherwise ($X$ is empty) the statement is ignored,
  - $\text{nil}(X)$ empties stack $X$,
  - if $\text{top}(X) \equiv a$ [P] executes the body $P$ if the top symbol on stack $X$ is identical to letter $a$, otherwise the conditional statement is ignored,

- loop statements foreach $X$ [P] are executed call-by-value, that is, first a local copy $U$ of $X$ is allocated and $P$ is altered to $P'$ by simultaneously replacing each “free occurrence”\(^1\) of $X$ in $P$ (appearing as if $\text{top}(X) \equiv a$ [Q]) with $U$; then one executes the sequence

$$P'; \text{pop}(U); \ldots; P'; \text{pop}(U) \quad (|X| \text{ times}).$$

Thus, when executing a loop foreach $X$ [P] the contents of the control stack $X$ is saved while providing access to each symbol on $X$.

We say that a stack program $P$ computes a function $f : (\Sigma^*)^n \to \Sigma^*$ if $P$ has an output variable $O$ and input variables $X_{i_1}, \ldots, X_{i_l}$ among stacks $X_1, \ldots, X_m$ such that for all $w_1, \ldots, w_n \in \Sigma^*$,

$$\{X_{i_1} = w_{i_1}, \ldots, X_{i_l} = w_{i_l}\} P \{O = f(w_1, \ldots, w_n)\}$$

often abbreviated by $\{\vec{X} = \vec{w}\} P \{O = f(\vec{w})\}$. Note that $O$ may occur among $X_{i_1}, \ldots, X_{i_l}$.

Throughout the paper, when speaking of a “subprogram” of another program $P$ we refer to its standard meaning, that is, any substring of $P$ which itself is a program.

4.2. The measure $\mu$ on stack programs

In the analysis of the computational complexity of stack programs $P$, the interplay of two kinds of variables will play a major role: updatable variables and control variables, constituting the sets $U(P)$ and $C(P)$.

$$U(P) := \{X \mid P \text{ contains an imperative } \text{push}(a, X)\}$$

$$C(P) := \{X \mid P \text{ contains a loop } \text{foreach } X [Q] \text{ with } U(Q) \neq \emptyset\}$$

Definition 4.2.1 (Control). Let $P$ be a stack program. The relation control in $P$, denoted $\prec_P$, is defined as the transitive closure of the following binary relation $\prec$ on $V(P)$:

$$X \prec_P Y :\iff P \text{ contains a loop } \text{foreach } X [Q] \text{ such that } Y \in U(Q).$$

We say that $X$ controls $Y$ in $P$ if $X \prec_P Y$, that is, there exist variables $X \equiv X_1, X_2, \ldots, X_l \equiv Y$ such that $X_1 \prec_P X_2 \prec_P \ldots \prec_P X_{l-1} \prec_P X_l$.

\(^1\)An occurrence of $X$ in $P$ is free if it does not appear in the body $Q$ of a subprogram foreach $X$ [Q] of $P$. 
Definition 4.2.2 (The \(\mu\)-measure). The \(\mu\)-measure of a stack program \(P\), denoted by \(\mu(P)\), is inductively defined as follows:

- \(\mu(\text{imp}) := 0\) for every imperative \(\text{imp}\).
- \(\mu(\text{if top}(X) \equiv a \{Q\}) := \mu(Q)\)
- \(\mu(Q_1; Q_2) := \max\{\mu(Q_1), \mu(Q_2)\}\)
- If \(P\) is a loop \(\text{foreach} X \{Q\}\), then
  \[
  \mu(P) := \begin{cases} 
  \mu(Q) + 1 & \text{if } Q \text{ is a sequence with a top circle} \\
  \mu(Q) & \text{else}
  \end{cases}
  \]

where \(Q \equiv Q_1; \ldots; Q_l\) has a top circle if there exists a component \(Q_i\) with \(\mu(Q_i) = \mu(Q)\) such that some \(Y\) controls some \(Z\) in \(Q_i\), and \(Z\) controls \(Y\) in \(Q_1; \ldots; Q_{i-1}; Q_{i+1}; \ldots; Q_l\).

We say that a stack program \(P\) has \(\mu\)-measure \(n\) if \(\mu(P) = n\).

As pointed out above, the polynomial-time computable functions will be characterised as the functions computable by a stack program where each body of a loop is circle-free, that is, has no top circle.

4.3. The Bounding Theorem for stack programs

In this section we will show that every function \(f\) computed by a stack program of \(\mu\)-measure \(n\) has a length bound \(b \in \mathbb{E}^{n+2}\), that is, \(|f(\bar{w})| \leq b(|\bar{w}|)\) for all \(\bar{w}\). It suffices to show this “bounding theorem” for a subclass of stack programs, called core programs. The latter comprise those stack manipulations which do contribute to computational complexity. The base case is treated separately, showing that core programs of \(\mu\)-measure 0 compute only polynomially length-bounded functions. For the general case, we show that every core program \(P\) of \(\mu\)-measure \(n+1\) has a “length bound” \(P'\) of \(\mu\)-measure \(n+1\). The structure of \(P'\) we call flattened out will be such that by a straightforward inductive argument we obtain that every function computed by \(P'\) has a length bound in \(\mathbb{E}^{n+2}\).

Definition 4.3.1 (Core programs). Core programs are stack programs built from imperatives \(\text{push}(a, X)\) by sequencing and loop statements.

Note 4.3.2. The chosen call-by-value semantics of loop statements ensures that core programs are length-monotonic, that is, if \(P\) is a core program with variables \(\bar{X}\), then

- if \(\bar{X} = \bar{w}\) \(P\) \(\{\bar{X} = \bar{u}\}\), then \(|\bar{w}| \leq |\bar{u}|\) (component-wise), and
- if \(|\bar{w}| \leq |\bar{w}'|\) and \(\bar{X} = \bar{w}\) \(P\) \(\{\bar{X} = \bar{u}\}\) and \(\bar{X} = \bar{w}'\) \(P\) \(\{\bar{X} = \bar{u}'\}\), then \(|\bar{u}| \leq |\bar{u}'|\).

Thus, functions computable by core programs are length-monotonic, too.

Lemma 4.3.3 (Measure Zero). For core programs \(P := \text{foreach} X \{Q\}\) with \(\mu(P) = 0\), \(\rightarrow\) is irreflexive.

Proof. By induction on the structure of core programs \(P := \text{foreach} X \{Q\}\) of \(\mu\)-measure 0. The statement is obvious if \(Q\) is an imperative \(\text{push}(a, Y)\). If \(Q\) is of the form \(\text{foreach} Y \{R\}\), the statement follows from the induction hypothesis on \(Q\) and \(X \notin \mathcal{U}(Q)\). Finally, if \(Q\) is a sequence \(Q_1; \ldots; Q_n\), then by the induction hypothesis on each component \(Q_i\), no \(Y\) controls \(Y\) in \(Q_i\). Therefore, if some \(Y\) controlled \(Y\) in \(Q_i\), then \(Y\) controlled some \(Z \neq Y\) in some \(Q_j\), and \(Z\) controlled \(Y\) in the context \(Q_1; \ldots; Q_{j-1}; Q_{j+1}; \ldots; Q_n\). Hence \(Q\) would have a top circle, contradicting the hypothesis \(\mu(P) = 0\). \(\square\)
Lemma 4.3.4 (Irreflexive Bounding). Let $\mathcal{P}$ be a core program with irreflexive $\xrightarrow{\bowtie}$. Let $\mathcal{P}$ have variables among $\vec{X} := x_1, \ldots, x_n$, and for $i = 1, \ldots, n$ let $V^i$ denote the list of those variables $x_j$ which control $x_i$ in $\mathcal{P}$. Then there are polynomials $p_1(V^1), \ldots, p_n(V^n)$ such that for all $\vec{w} := w_1, \ldots, w_n$,

$$\{ \vec{X} = \vec{w} \} \mathcal{P} \{ |x_i| \leq |w_i| + p_i(|\vec{w}|) \} \quad \text{for } i = 1, \ldots, n$$

where $\vec{w}^i$ results from $\vec{w}$ by selecting those $w_j$ for which $X_j$ is in $V^i$.

Proof. By induction on the structure of core programs $\mathcal{P}$ with irreflexive $\xrightarrow{\bowtie}$. In the base case where $\mathcal{P}$ is of the form $\text{push}(a, X_1)$, we know $V^1 = \emptyset$ and hence $p_1 := 1$ will do.

Step case $\mathcal{P}$ is a sequence $\mathcal{P}_1 ; \mathcal{P}_2$. The induction hypothesis yields polynomials $q_1(V^1), \ldots, q_n(V^n)$ for $\mathcal{P}_1$, and polynomials $r_1(V^1), \ldots, r_n(V^n)$ for $\mathcal{P}_2$. Now fix any $i$ among $1, \ldots, n$, and suppose that $X_{i_1}, \ldots, X_{i_l}$ are the variables which control $X_i$ in $\mathcal{P}$. Then one easily verifies that

$$\{ \vec{X} = \vec{w} \} \mathcal{P} \{ |x_i| \leq |w_i| + q_i(|\vec{w}|) \} \quad \text{for } i = 1, \ldots, n.$$ 

As $\xrightarrow{\bowtie}$ is irreflexive, then so is $\xrightarrow{\bowtie}$ via $\mathcal{Q}$. From this it follows that $\xrightarrow{\bowtie}$ defines a strict partial order on $\vec{X}$. Therefore, we can proceed by side induction on this strict partial order showing that one can find polynomials $q_1(m, V^1), \ldots, q_n(m, V^n)$ such that for all $m$, $\vec{w}$,

$$\{ \vec{X} = \vec{w} \} \mathcal{Q}^m \{ |x_i| \leq |w_i| + q_i(m, |\vec{w}|) \} \quad \text{for } i = 1, \ldots, n$$

where $Q^m$ denotes the sequence $Q; \ldots; Q (m \text{ times})$. Note that (2) implies the statement of the lemma for the current case $\mathcal{P} := \text{foreach } X_j [Q]$. To see this, if $X_i \notin U(Q)$ then $V_i = \emptyset$ and the execution of $Q$ does not alter the contents of $X_i$, hence $p_i := 0$ will do. Otherwise if $X_i \in U(Q)$, then $X_j \in V_i$ and (2) gives

$$\{ \vec{X} = \vec{w} \} \mathcal{P} \{ |x_i| \leq |w_i| + q_i(|\vec{w}|) \} \quad \text{where } w_j \in \vec{w}.$$

For the proof of (2), fix any $i \in \{1, \ldots, n\}$. If $V_i = \emptyset$ then $p_i$ in (1) is a constant. From this it follows

$$\{ \vec{X} = \vec{w} \} \mathcal{P} \{ |x_i| \leq |w_i| + m \cdot p_i \} \quad \text{so consider the case } V_i = X_{i_1}, \ldots, X_{i_l} \text{ with } l \geq 1.$$ 

The side induction hypothesis yields polynomials $r_i(m(V_i^1), \ldots, r_i(m, V_i^l))$ such that for all $m$, $\vec{w}$,

$$\{ \vec{X} = \vec{w} \} \mathcal{Q}^m \{ |x_{i_j}| \leq |w_{i_j}| + r_i(m, |\vec{w}|) \} \quad \text{for } j = 1, \ldots, l$$

where $V_i^j$ denotes the variables which control $X_j$ in $Q$. Observe that $X_{i_j} \notin V_i^j$ for $j = 1, \ldots, l$ and $X_i \notin V_i^1 \cup \ldots \cup V_i^l \subseteq V_i$. Hence it suffices to prove by induction on $m$ that for all $m$, $\vec{w}$,

$$\{ \vec{X} = \vec{w} \} \mathcal{Q}^m \{ |x_i| \leq |w_i| + m \cdot p_i(|w_i| + r_i(m, |\vec{w}|)) \}.$$ 

The base case $m = 0$ is obviously true. As for the step case $m \rightarrow m + 1$, assume that

$$\{ \vec{X} = \vec{w} \} \mathcal{Q}^m \{ X_{i_1} = u_{i_1}, \ldots, X_{i_l} = u_{i_l}, X_i = v_i, \ldots \} \mathcal{Q} \{ X_i = v_i^* \}.$$ 

Utilising the various induction hypotheses at hand, we obtain the following estimations:

$$|v_i^*| \leq |v_i| + p_i(|u_{i_1}|, \ldots, |u_{i_l}|) \quad \text{by the main I.H. (1)}$$

$$|v_i| \leq |w_i| + m \cdot p_i(|w_{i_j}| + r_i(m, |\vec{w}_{i_j}|), \ldots) \quad \text{by the I.H. for } m$$

$$|u_{i_j}| \leq |w_{i_j}| + r_i(m, |\vec{w}_{i_j}|) \quad \text{for } j = 1, \ldots, l \quad \text{by the side I.H. (3)}$$

Combining these estimations and using monotonicity of polynomials, one easily obtains the required estimation \(|v_i^*| \leq |w_i| + (m + 1) \cdot p_i(\ldots, \max|w_j|, r_i(m + 1, |w_j|), \ldots)|. This concludes the proof of \((4)\) and thus the proof for the current case \(P \equiv \text{foreach } X_j [Q]. \)

**Corollary 4.3.5 (BaseBounding).** For every core program \(P\) with \(\mu(P) = 0\) and variables \(X := X_1, \ldots, X_n\) one can find polynomials \(p_1(X), \ldots, p_n(X)\) such that for all \(\vec{w} := w_1, \ldots, w_n,\)

\[
\{ \vec{x} = \vec{w} \} P \{ |x_1| \leq p_1(|\vec{w}|), \ldots, |x_n| \leq p_n(|\vec{w}|) \}.
\]

In particular, every function computed by \(P\) is polynomially length-bounded, too.

**Proof.** The statement of the corollary follows from Irreflexive Bounding (4.3.4), Measure Zero (4.3.3), Note 4.1.2, and the fact that polynomials are closed under composition.

We are now going to treat the general case in the proof of the Bounding Theorem mentioned above. For this purpose, we first define what we mean by saying that one core program is a length bound on another, and how flattened out core programs look like.

In the following, let \(n\) be a fixed, but arbitrary natural number.

**Definition 4.3.6 (Lengthbound).** For stack programs \(P, Q\) such that \(\mathcal{V}(P) = \{X_1, \ldots, X_k\}\) is contained in \(\mathcal{V}(Q) = \mathcal{V}(P) \cup \{Y_1, \ldots, Y_l\}\) we say that \(Q\) is a length bound on \(P\), denoted \(P \ll Q\), if

\[
\{ \vec{x} = \vec{w} \} P \{ \vec{x} = \vec{v} \} \text{ and } \{ \vec{x} = \vec{w}, \vec{y} = \vec{u} \} Q \{ \vec{x} = \vec{v} \} \text{ implies } |\vec{w}| \leq |\vec{v}|.
\]

**Definition 4.3.7 (Simpleloops and flattenedout coreprograms).**

- A loop \(\text{foreach } X [Q]\) of \(\mu\)-measure \(n + 1\) is simple if \(Q\) has \(\mu\)-measure \(n\).
- A core program \(P := P_1; \ldots; P_k\) of \(\mu\)-measure \(n + 1\) is flattened out if each component \(P_i\) is either a simple loop or else \(\mu(P_i) \leq n\).

Given a core program \(P\) of \(\mu\)-measure \(n + 1\), we want to construct a flattened out core program \(P'\) of \(\mu\)-measure \(n + 1\) such that \(P'\) is a length bound on \(P\). To succeed in that goal, it suffices to transform, step by step, certain occurrences of non-simple loops in \(P\). That motivates the next definition where we make use of the standard notion of nesting depth \(\deg(P)\) for core programs \(P\), that is, \(\deg(\text{push} (a, X)) := 0, \deg(P_1; P_2) := \max\{\deg(P_1), \deg(P_2)\},\) and \(\deg(\text{foreach } X [Q]) := 1 + \deg(Q)\).

**Definition 4.3.8 (Rank).** The rank of a core program \(P\), denoted \(\text{rk}(P)\), is inductively defined by:

- \(\text{rk}(\text{push} (a, X)) := 0\) for every letter \(a \in \Sigma\) and variable \(X\).
- \(\text{rk}(P_1; P_2) := \max\{\text{rk}(P_1), \text{rk}(P_2)\}\).
- If \(P\) is a loop \(\text{foreach } X [Q]\), then

\[
\text{rk}(P) := \begin{cases} 
0 & \text{if } P \text{ is a simple loop or } \mu(P) \leq n \\
1 + \text{rk}(Q) & \text{if } Q \text{ is a loop with } \mu(Q) = n + 1 \\
1 + \sum_{i \leq k} \deg(Q_i) & \text{if } Q \text{ is a sequence } Q_0, \ldots; Q_k \text{ without top circle and } \mu(Q) = n + 1.
\end{cases}
\]
Lemma 4.3.9 (Rank Zero). Every core program \( P \) of \( \mu \)-measure \( n + 1 \) and rank 0 is flattened out.

Proof. By induction on the structure of core programs \( P \) of \( \mu \)-measure \( n + 1 \) and rank 0. If \( P \) is a sequence \( P_1; P_2 \) then both components \( P_i \) have rank 0, and at least one has \( \mu \)-measure \( n + 1 \). Hence the claim follows from the induction hypothesis on the components of \( \mu \)-measure \( n + 1 \). If \( P \) is a loop of \( \mu \)-measure \( n + 1 \) and rank 0, then this loop is simple by definition, hence \( P \) is flattened out.

Lemma 4.3.10 (Rank Reduction). Every core program \( P :≡ \text{foreach} \ X \ [Q] \) with \( \mu(P) = n + 1 \) and rank \( > 0 \) has a core program \( P' \) satisfying \( P ≪ P' \), \( \mu(P') = n + 1 \) and \( \text{rk}(P') < \text{rk}(P) \).

Proof. Let \( P :≡ \text{foreach} \ X \ [Q] \) be an arbitrary core program of \( \mu \)-measure \( n + 1 \) and rank \( > 0 \). According to definition 4.3.8 and Note 4.1.2, we distinguish two cases on \( Q \).

Case \( Q \) is a loop \( \text{foreach} \ Y \ [R] \) with \( \mu(P) = \mu(Q) = n + 1 \). In that case we define \( P' \) by

\[
P' :≡ \text{foreach} \ X \ [\text{foreach} \ Y \ [\text{push}(a, Z)]]; \text{foreach} \ Z \ [R]
\]

for some new variable \( Z \) and arbitrary letter \( a \). Obviously, \( \mu(P') = \mu(P) \) and \( P ≪ P' \). As for \( \text{rk}(P') < \text{rk}(P) \), first observe that \( \text{rk}(P) = 1 + \text{rk}(Q) \) and \( \text{rk}(P') = \text{rk}(\text{foreach} Z \ [R]) \). Thus, we obtain \( \text{rk}(P) > \text{rk}(Q) = \text{rk}(\text{foreach} Y \ [R]) = \text{rk}(\text{foreach} Z \ [R]) = \text{rk}(P') \) as required.

Case \( Q \) is a sequence \( Q_0; \ldots ; Q_k \) without top circle, and \( \mu(Q_i) = \mu(Q) = n + 1 \) for some component \( Q_i :≡ \text{foreach} \ Y \ [R] \). In that case we define \( P' :≡ \text{foreach} \ X \ [P_1]; \text{foreach} \ Z \ [P_2] \) where for some new variable \( Z \) and any \( a \in \Sigma \) the core programs \( P_1, P_2 \) are defined by:

\[
P_1 :≡ Q_0; \ldots ; Q_{i-1}; \text{foreach} Y \ [\text{push}(a, Z)]; Q_{i+1}; \ldots ; Q_k
\]

\[
P_2 :≡ Q_0; \ldots ; Q_{i-1}; R; Q_{i+1}; \ldots ; Q_k
\]

In terms of \( P ≪ P' \), let \( \#R[P(\vec{w})] \) denote the number of times \( R \) is executed in a run of \( P \) on \( \vec{w} \). Then we conclude \( \#R[P(\vec{w})] ≪ \#\text{push}(a, Z)[P'(\vec{w}, v)] \), for otherwise some stack \( V ∈ U(R) \) controlled \( Y \) in \( Q_0; \ldots ; Q_{i-1}; Q_{i+1}; \ldots ; Q_k \), contradicting the hypothesis that \( Q \) has no top circle. Thus \( \{ \vec{X} = \vec{w}, Z = v \mid \text{foreach} X \ [P_1] \} \{ |Z| ≥ \#R[P(\vec{w})] \} \), and that implies \( P ≪ P' \) by monotonicity of core programs.

It remains to show \( \mu(P') = n + 1 \) and \( \text{rk}(P') < \text{rk}(P) \). First observe that \( R \) contains a loop, as \( Q_i \) has \( \mu \)-measure \( n + 1 \). We distinguish two subcases.

Subcase \( Q_i \) is a simple loop, that is, \( \mu(Q_i) = 1 + \mu(R) \) and \( R \) is a sequence with top circle. First consider the case where \( Q_i \) is the only component of \( Q \) with \( \mu \)-measure \( n + 1 \). Then each component of \( P_1 \) is of \( \mu \)-measure \( n \), and \( P_2 \) is a sequence with a top circle. This implies \( \mu(P') = \mu(\text{foreach} Z \ [P_2]) = n + 1 \), and thus \( \text{foreach} Z \ [P_2] \) is a simple loop, resulting into \( \text{rk}(P') = \text{rk}(\text{foreach} X \ [P_1]) \). Now observe that either \( P_1 \) has a top circle, in which case \( \text{foreach} Z \ [P_1] \) is a simple loop, or else \( \mu(P_1) ≤ n \). In either case, we obtain \( \text{rk}(P') = 0 < \text{rk}(P) \). So consider the case where \( \mu(Q_j) = n + 1 \) for some \( j \neq i \). Then both \( P_1 \) and \( P_2 \) are sequences without top circle, implying \( \mu(P') = \mu(\text{foreach} Z \ [P_2]) = n + 1 \) and, as \( R \) contains a loop, \( \text{rk}(P') = \text{rk}(\text{foreach} Z \ [P_2]) \). Now \( \text{deg}(R) < \text{deg}(Q_i) \) implies \( \text{rk}(\text{foreach} Z \ [P_2]) < \text{rk}(P) \), concluding the current subcase.

Subcase \( Q_i \) is a not a simple loop, hence \( \mu(Q_i) = \mu(R) \) and \( R \) is either a loop or a sequence without top circle. In either case, \( P_2 \) has no top circle, implying \( \mu(P') = \mu(\text{foreach} Z \ [P_2]) = n + 1 \). Furthermore, as \( R \) contains a loop, we have \( \text{deg}(Q_i) > \text{deg}(R) ≥ \text{deg}(\text{foreach} Y \ [\text{push}(a, Z)]) \) and thus \( \text{rk}(P') < \text{rk}(P) \), concluding the proof of the lemma.
Lemma 4.3.11 (Flattening). For every core program $P$ of $\mu$-measure $n + 1$ one can find a flattened out core program $P'$ of $\mu$-measure $n + 1$ satisfying $P \ll P'$.

Proof. The statement follows from Note 4.1.2, Rank Reduction (4.3.10) and Rank Zero (4.3.9). □

As pointed out above, the Flattening Lemma establishes the following Bounding Theorem.

Theorem 4.3.12 (Bounding). Every function $f$ computed by a stack program of $\mu$-measure $n$ has a length bound $b \in \mathcal{E}^{n+2}$ satisfying $|f(\vec{w})| \leq b(|\vec{w}|)$.

Proof. It suffices to prove the statement of the theorem for core programs only, because for every stack program $P$ one can find a core program $P^*$ such that $\mu(P) = \mu(P^*)$ and $Q \ll P^*$ for every subprogram $Q$ of $P$. Just let $P^*$ result from $P$ by recursively replacing all occurrences of imperatives $\text{nil}(X)$ or $\text{pop}(X)$ with $\text{foreach } X \{ \text{push}(b, \mathcal{V}) \}$, and all conditional statements $\text{if top(X) \equiv a \{ Q \} with the sequence}$ $\text{foreach } X \{ \text{push}(b, \mathcal{V}) \} ; Q^*$, for some new variable $\mathcal{V}$ and some letter $b \in \Sigma$.

We proceed by induction on $n$ showing the statement of the theorem for core programs. The base case $n = 0$ is treated in Base Bounding (4.3.5). For the step case $n \rightarrow n + 1$, consider any core program $P$ of $\mu$-measure $n + 1$. We apply Flattening (4.3.11) to obtain a core program $P'$ of the form $P_1 ; \ldots ; P_k$ where each component $P_i$ is either a simple loop or else $\mu(P_i) \leq n$, and such that $P \ll P'$ and $\mu(P') = n + 1$. Thus, by the induction hypothesis and by closure of $\mathcal{E}^{n+3}$ under composition, it suffices to show that every function computed by a simple loop $P_i$ has a length bound in $\mathcal{E}^{n+3}$.

Let $P_i := \text{foreach } X \{ Q \}$ be any such simple loop. Hence $\mu(Q) = n$ and by the induction hypothesis each function $h_j$ computed by $Q$ has a length bound $b_j \in \mathcal{E}^{n+2}$. We choose a number $c > 0$ such that $b_j(\vec{x}) \leq E^{(c)}_{n+1}(\max(\vec{x}))$ for each bound $b_j$. Let $f_1$ be any function computed by $P_i$. Then $f_1$, possibly together with other functions $f_2, \ldots, f_m$ computed by $P_i$, can be defined by simultaneous string recursion from functions computed by $Q$, that is,

\[
\begin{align*}
  f_k(\varepsilon, \vec{w}) &= w_k, \\
  f_k(v \cdot a, \vec{w}) &= h_k(v, \vec{w}, f_1(v, \vec{w}), \ldots, f_m(v, \vec{w})) \quad \text{for } k = 1, \ldots, m.
\end{align*}
\]

It follows by induction on $|v|$ that $|f_k(v, \vec{w})| \leq E^{(c)}_{n+1}(\max(|v, \vec{w}|))$. As $E^{(t)}_{n+1}(x) \leq E_{n+2}(x + t)$ and $\max, + \in \mathcal{E}^2$, we therefore obtain a length bound on $f_1$ in $\mathcal{E}^{n+3}$.

4.4. The Characterisation Theorem for stack programs

In this section it is shown that stack programs of $\mu$-measure $n$ compute exactly the functions computable by a Turing machine in time bounded by a function in $\mathcal{E}^{n+2}$. Furthermore, a simulation of stack programs over an arbitrary alphabet by such over a binary alphabet, preserving the measure $\mu$, is presented. Thus, stack programs of $\mu$-measure $0$ compute precisely the polynomial-time computable functions.

Given any alphabet $\Sigma$, a function over $\Sigma^*$ is any $k$-ary function $f : \Sigma^* \times \ldots \times \Sigma^* \rightarrow \Sigma^*$.

Definition 4.4.1 ($\mathcal{L}^n$). For $n \geq 0$ let $\mathcal{L}^n$ denote the class of all functions $f$ over $\Sigma^*$ (for some alphabet $\Sigma$) which can be computed by a stack program of $\mu$-measure $n$.

Definition 4.4.2 ($\mathcal{G}^n$). For $n \geq 0$ let $\mathcal{G}^n$ denote the class of all functions $f$ over $\Sigma^*$ (for some alphabet $\Sigma$) which can be computed by a Turing machine in time $b(|\vec{w}|)$ for some $b \in \mathcal{E}^n$. 
Lemma 4.4.3 \((E_{n+1}\text{-Simulation})\). For every \(n \geq 0\) one can find a sequence \(\text{LE}[n+1]\) with a top circle such that \(\mu(\text{LE}[n+1]) = n\) and \(\{Y = w\} \text{LE}[n+1] \{ |Y| = E_{n+1}(|w|) \}\).

Proof. By induction on \(n\). The base case for \(E_1(x) = x^2 + 2\) is obvious. As for the step case, first recall that \(E_{n+2}(x) = E_{n+1}(\ldots E_{n+1}(2)\ldots)\) with \(x\) occurrences of \(E_{n+1}\). Using the induction hypothesis on \(n\), and for some new variable \(U\), we define \(\text{LE}[n+1]\) by:

\[
\begin{align*}
\text{LE}[n+2] & := \text{nil}(U); \text{foreach } Y [\text{push}(a,U)]; \\
& \quad \text{nil}(Y); \text{push}(a,Y); \text{push}(a,Y); \text{foreach } U [\text{LE}[n+1]].
\end{align*}
\]

Theorem 4.4.4 (Characterisation). For \(n \geq 0: \mathcal{L}^n = G^{n+2}\).

Proof. First we prove the inclusion “\(\subseteq\)”. Let \(\mathcal{P}\) be an arbitrary stack program of \(\mu\)-measure \(n\). Let \(\text{TIME}_\mathcal{P}(\vec{w})\) denote the number of steps in a run of \(\mathcal{P}\) on input \(\vec{w}\), where a step is the execution of an arbitrary imperative \(\text{imp}(X)\). Note that there is a polynomial \(\text{qtime}(n)\) such that each step \(\text{imp}(X)\) can be simulated on a Turing machine in time \(\text{qtime}(|X|)\). Now let \(V\) be any new variable, \(a \in \Sigma\) any letter, and let \(\mathcal{P}^\#\) result from \(\mathcal{P}\) by replacing each imperative \(\text{imp}\) with \(\text{imp}; \text{push}(a,V)\). Then the program \(\text{TIME}(\mathcal{P}) := \text{nil}(V); \mathcal{P}^\#\) has \(\mu\)-measure \(n\) and satisfies \(\{\vec{X} = \vec{w}\} \text{TIME}(\mathcal{P}) \{ |V| = \text{TIME}_\mathcal{P}(\vec{w}) \}\). Therefore we may apply Bounding (4.3.12) to obtain a length bound \(b \in \mathcal{E}^{n+2}\) satisfying

\[
\{\vec{X} = \vec{w}\} \text{TIME}(\mathcal{P}) \{ |V| \leq b(|\vec{w}|) \}.
\]

Hence there is a Turing machine which simulates \(\mathcal{P}\) on input \(\vec{w}\) in time \(\text{qtime}(b(|\vec{w}|)) \cdot b(|\vec{w}|)\), concluding the proof of the inclusion “\(\subseteq\)”. As for “\(\supseteq\)”, let \(M := (Q, \Gamma, \Sigma, q_0, \delta)\) be an arbitrary one-tape Turing machine running in time \(b(|w|)\) on input \(w\), for some \(b \in \mathcal{E}^{n+2}\). We show that the function \(f_M\) computed by \(M\) can be computed by a stack program \(\mathcal{P}\) over \(\Delta := Q \cup \Gamma \cup \{L, N, R\}\) of \(\mu\)-measure \(n\), where \(\Gamma := \{a_1, \ldots, a_k\}\). Assume that \(\delta\) consists of moves \(\text{move}_1, \ldots, \text{move}_t\) where

\[
\text{move}_i := (q_i, a_i, q_i', a_i', D_i)
\]

with \(D_i \in \{L, N, R\}\). The program \(\mathcal{P}\) of \(\mu\)-measure \(n\) satisfying \(\{X = w\} \mathcal{P} \{ \omega = f_M(w) \}\) uses stacks \(X, Y, Z, L, R, \ldots\) and will have the following form:

\[
\begin{align*}
\mathcal{P} & := \text{COMPUTE\text{-}TIME\text{-}BOUND}(Y); \\
& \quad \text{INITIALISE}(L, Z, R); \quad \text{(}\ast \text{ of } \mu\text{-measure } n \ast) \\
& \quad \text{foreach } Y [\text{SIMULATE\text{-}MOVES}]; \quad \text{(}\ast \text{ of } \mu\text{-measure } 0 \ast) \\
& \quad \text{OUTPUT}(R; \omega) \quad \text{(}\ast \text{ of } \mu\text{-measure } 0 \ast)
\end{align*}
\]

Let \(\text{INIT}\) denote the initial part \(\text{COMPUTE\text{-}TIME\text{-}BOUND}(Y); \text{INITIALISE}(L, Z, R)\) of \(\mathcal{P}\). Then the simulation of \(M\) will be such that for each configuration \(\alpha(q, a)\beta\) in a run of \(M\) on \(w\) obtained after \(m\) steps, that is, \(\text{init}_M(w) \with^m \alpha(q, a)\beta\), we have:

\[
\{X = w\} \text{INIT}; \text{SIMULATE\text{-}MOVES}^m \{L = \alpha, \text{reverse}(R) = a\beta, Z = q\}
\]

Recall that there is a constant \(c\) satisfying \(b(x) \leq E_{n+1}^c(x)\), and by \(E_{n+1}\text{-Simulation}\) (4.4.3) there is a stack program of \(\mu\)-measure \(n\) such that \(\{Y = w\} \text{LE}[n+1] \{ |Y| = E_{n+1}(|w|) \}\). Thus,

\[
\text{COMPUTE\text{-}TIME\text{-}BOUND}(Y) := \text{nil}(Y); \text{foreach } X [\text{push}(a,Y)]; \text{LE}[n+1]; \ldots; \text{LE}[n+1]
\]
The Characterisation Theorem for stack programs

According to (*), we initialise $L$, $Z$, $R$ as follows:

$$INITIALISE(L,Z,R) \equiv \text{nil}(L); \text{set}(Z,q_0); \text{REVERSE}(X;R)$$

where REVERSE is a stack program satisfying $\{X = w\} \text{REVERSE}(X;R) \{X = w, R = \text{reverse}(w)\}$.

As for SIMULATE-MOVES, we use several short forms to facilitate reading. First note that conditionals if $X \equiv \varepsilon [Q]$ and if $X \not\equiv \varepsilon [Q]$ of $\mu$-measure $\mu(Q)$ can be defined by

if $X \equiv \varepsilon [Q] : \equiv \text{nil}(U); \text{push}(a,U); \text{foreach} X[\text{pop}(U)]; \text{iftop}(U) \equiv a[Q]$

if $X \not\equiv \varepsilon [Q] : \equiv \text{nil}(U); \text{push}(a,U); \text{foreach} X[\text{pop}(U)]; \text{if} U \equiv \varepsilon [Q]$

where $U$ is some new variable, and $a$ is any letter in the tape alphabet $\Gamma$. Similarly easy, one defines conditionals if $\text{top}(R) \in \Sigma [Q]$ and if $\text{top}(R) \in \Delta \setminus \Sigma [Q]$ of $\mu$-measure $\mu(Q)$. Finally, we use

$$\text{set}(U,a) \quad \text{for} \quad \text{nil}(U); \text{push}(a,U)$$

$$\text{settop}(U,a) \quad \text{for} \quad \text{pop}(U); \text{push}(a,U)$$

$$\text{push}(\text{top}(L),R) \quad \text{for} \quad \text{if} \text{top}(L) \equiv a_1 [\text{push}(a_1,R)]; \ldots; \text{if} \text{top}(L) \equiv a_k [\text{push}(a_k,R)].$$

SIMULATE-MOVES is of the form $\text{MOVE}_1; \ldots; \text{MOVE}_k$ where $\text{MOVE}_i$ simulates move$_i$. According to $D_i = R, L, N$ in $\text{move}_i = (q_i, a_i, q'_i, a'_i, D_i)$, there are three cases for $\text{MOVE}_i$:

$$\text{(R)} \quad \text{if} \text{top}(Z) \equiv q_i \left[ \text{if} \text{top}(R) \equiv a_i \left[ \text{push}(a'_i,L); \text{set}(Z,q'_i)\right] \right]; \text{pop}(R); \text{if} R \equiv \varepsilon \left[ \text{push}(B,R) \right]$$

$$\text{(L)} \quad \text{if} \text{top}(Z) \equiv q_i \left[ \text{if} \text{top}(R) \equiv a_i \left[ \text{settop}(R,a'_i); \text{set}(Z,q'_i)\right] \right]; \text{if} L \equiv \varepsilon \left[ \text{push}(B,R) \right]; \text{if} L \not\equiv \varepsilon \left[ \text{push}(\text{top}(L),R); \text{pop}(L) \right]$$

$$\text{(N)} \quad \text{settop}(R,a'_i); \text{set}(Z,q'_i)$$

It remains to implement $\text{OUTPUT}(R;O)$ which reads out of stack $R$ the result $O = f_M(w)$, that is, the maximal initial segment of $\text{reverse}(R)$ being a word over $\Sigma$. We do this as follows:

$$\text{OUTPUT}(R;O) : \equiv \text{nil}(O); \text{set}(Z,a); \text{foreach} R \left[ \text{if} \text{top}(R) \in \Delta \setminus \Sigma \left[ \text{nil}(Z) \right]; \text{if} \text{top}(R) \in \Sigma \left[ \text{if} \text{top}(Z) \equiv a \left[ \text{push}(\text{top}(R),O) \right] \right] \right]$$

This completes the proof of the Characterisation Theorem. $\square$

It is worthwhile to emphasise that the proof of $G^{n+2} \subseteq \mathcal{L}^n$ would fail for $n = 0$ if the development were based on a traditional measure, such as the nesting depth $\text{deg}$. For given a Turing machine which runs in polynomial time, there is no implementation of $\text{COMPUTE\text{-}TIME\text{-}BOUND}$ of $\text{deg}$-measure 0. This would not even work for $\text{deg}$-measure 1, and for $\text{deg}$-measure $\geq 2$ one cannot separate polynomial from exponential running time.

Observe that the proof of $\mathcal{L}^n \subseteq G^{n+2}$ actually shows that every stack program of $\mu$-measure $n$ can be simulated by a Turing machine with running time bounded in $E^{n+2}$. Furthermore, if a Turing machine $M$ with running time bounded in $E^{n+2}$ is a decision procedure, one obtains a stack simulation of $M$ of $\mu$-measure $n$ by replacing in $P$ above the part $\text{OUTPUT}(R;O) \text{ with \ } \text{nil}(O); \text{push}(\text{top}(Z),O)$. This gives the following corollary.
Corollary 4.4.5 (Main). Stack programs of $\mu$-measure $n$ simulate exactly Turing machines which run in time bounded by a function in $E^{n+2}$.

In the proof above, given a Turing machine $M$ with running time bounded in $E^{n+2}$, we have chosen a convenient alphabet (depending on $M$) in order to simulate $M$ by a stack program of $\mu$-measure $n$. Rounding off this section, we prove that stack programs over an arbitrary alphabet can be simulated by such over a binary alphabet, preserving the $\mu$-measure. Thus, in particular, stack programs of $\mu$-measure 0 compute exactly the polynomial-time computable functions.

Lemma 4.4.6 (Binary Simulation). Every stack program $P$ over $\Sigma := \{a_1, \ldots, a_l\}, l \geq 3$, can be simulated by a stack program $\text{sim}(P)$ over $\{0, 1\}$, preserving the $\mu$-measure, that is, $\mu(\text{sim}(P)) = \mu(P)$, and if $P$ on input $\vec{w}$ outputs $v$ then $\text{sim}(P)$ on input $\langle \vec{w} \rangle$ outputs $\langle v \rangle$, where for $i = 1, \ldots, L := |\text{bin}(l)|$

$$\langle a_i \rangle := b_1^i \ldots b_L^i := 0^k \text{bin}(i) \text{ such that } k + |\text{bin}(i)| = L$$

and $\langle c_1 \ldots c_r \rangle := \langle c_1 \rangle \ldots \langle c_r \rangle$ for words $c_1 \ldots c_r$ over $\Sigma$.

Proof. By induction on the structure of stack programs $P$, where the base cases are obvious:

$$\text{sim}(\text{nil}(X)) := \text{nil}(X)$$
$$\text{sim}(\text{pop}(X)) := \text{pop}(X); \ldots; \text{pop}(X) \quad (L \text{ times})$$
$$\text{sim}(\text{push}(a_i, X)) := \text{push}(b_1^i, X); \ldots; \text{push}(b_L^i, X)$$

Step case $P$ is if $\text{top}(X) \equiv a_i [Q]$. The induction hypothesis yields a simulation $\text{sim}(Q)$ of $Q$. For new variables $U, V$ we define $\text{sim}(P)$ by if $\text{TOP}(X) \equiv L b_1^i \ldots b_L^i \text{ sim}(Q)$, that is:

$$\text{nil}(U); \text{set}(V, 0);$$
$$\text{if top}(X) \equiv b_L^i \text{ [pop}(X); \text{push}(b_L^i, U);$$
$$\vdots$$
$$\text{if top}(X) \equiv b_1^i \text{ [pop}(X); \text{push}(b_1^i, X); \ldots; \text{push}(b_L^i, X); \text{sim}(Q); \text{set}(V, 1)] \ldots$$

if $\text{top}(V) \equiv 0$ [foreach $U$ [if $\text{top}(U) \equiv 0$ [push(0, X)];
$$\text{if top}(U) \equiv 1$$ [

Observe that stack $V$ is set to 1 if and only if the top $L$ letters in $X$ match the binary code $\langle a_i \rangle$. Thus, if that matching should fail at some point, then the letters chopped off and stored in $U$ so far are restored on $X$.

Step case $P$ is a sequence $P_1; P_2$. Here we define $\text{sim}(P)$ as the sequence $\text{sim}(P_1); \text{sim}(P_2)$.

Step case $P$ is a loop foreach $X [Q]$. The induction hypothesis yields a simulation $\text{sim}(Q)$ of $Q$. For new variables $U, V, Y$ we then define $\text{sim}(P)$ by:

$$\text{sim}(P) := \text{nil}(Y); \text{nil}(V);$$
$$\text{foreach } X [\text{if } |V| < L [\text{push}(1, V); \text{push}(\text{top}(X), Y)];$$
$$\text{if } |V| \geq L [\text{sim}(Q)^*; \text{nil}(Y); \text{nil}(V)]$$

where $\text{sim}(Q)^*$ results from $\text{sim}(Q)$ by replacing recursively each occurrence of a conditional statement $\text{sim}(\text{if top}(X) \equiv a_j [R])$ with conditional if $\text{TOP}(Y) \equiv L b_1^j \ldots b_L^j \text{ sim}(R)^*$ (see above), and
where the conditionals $\text{if } |V| < L [...] \text{ and } \text{if } |V| \geq L [...] \text{ are implemented as follows:}$

$$\text{if } |V| < L [...] \equiv \text{SET}(U, 1^L); \text{foreach } V \{ \text{pop}(U) \}; \text{if } \text{top}(U) \equiv 1 [...]$$

$$\text{if } |V| \geq L [...] \equiv \text{SET}(U, 1^{L-1}); \text{foreach } U \{ \text{pop}(V) \}; \text{if } \text{top}(V) \equiv 1 [...]$$

### 4.5. Loop programs and the Grzegorczyk hierarchy

In this section we define another simple loop language and apply the methods developed so far so as to obtain a characterisation of the Grzegorczyk hierarchy at and above linear-space level. Programs written in this language are called *loop programs*. They are built from primitive instructions $\text{nil}(X)$, $\text{suc}(X)$ and $\text{pred}(X)$ by sequence and loop statements, where $X$ is a variable. We assume an infinite supply of variables $X, Y, Z, O, U, V$, possibly with subscripts. Intuitively, variables serve as registers, each holding a natural number which can be manipulated by running a loop program. As with stack programs, we will define a measure $\mu$ on loop programs and relate it to the Grzegorczyk hierarchy by showing that loop programs of $\mu$-measure $n$ compute exactly the functions in $E^{n+2}$.

**Definition 4.5.1 (Loop programs).** Loop programs $P$ are inductively defined as follows:

- Every imperative among $\text{nil}(X), \text{suc}(X), \text{pred}(X)$ is a loop program.

- If $P_1, P_2$ are loop programs, then so is the *sequence* statement $P_1; P_2$.

- If $P$ is a loop program with no occurrence of $\text{suc}(X), \text{nil}(X)$ or $\text{pred}(X)$, then so is the *loop* statement $\text{loop } X \{ P \}$.

Again we use $V(P)$ to denote the set of variables occurring in $P$.

**Note 4.5.2.** Any loop program can be written uniquely in the form $P_1; \ldots; P_k$ such that each $P_i$ is either a loop or an imperative, and where $k = 1$ whenever $P$ is an imperative or a loop.

Loop programs have a standard semantics, e.g. like Pascal or C programs. The imperative $\text{nil}(X)$ sets register $X$ to zero. The imperative $\text{suc}(X)$ increments the number stored in $X$ by one, while $\text{pred}(X)$ decrements any nonzero number stored in $X$ by one. Imperatives and loops in a sequence are executed one by one from the left to the right. The meaning of a loop statement $\text{loop } X \{ P \}$ is that $P$ is executed $x$ times whenever the number $x$ is stored in $X$. Observe that $x$ is not updated during the execution of the loop statement $\text{loop } X \{ P \}$.

Obviously, the imperatives $\text{nil}(X), \text{pred}(X)$ do not contribute to the growth rate of functions computed by loop programs. Therefore, for loop programs $P$ we define:

$$U(P) := \{ X \mid P \text{ contains an imperative } \text{suc}(X) \}$$

$$C(P) := \{ X \mid P \text{ contains a loop } \text{loop } X \{ Q \} \text{ with } U(Q) \neq \emptyset \}$$

Now the concept of “control” for stack programs (cf. 4.2.1) passes on to loop programs. As well, the *measure* $\mu$ on loop programs is defined as on stack programs, that is, $\mu(\text{imp}) := 0$ for every imperative $\text{imp}$, $\mu(Q_1; Q_2) := \max\{\mu(Q_1), \mu(Q_2)\}$, and

$$\mu(\text{loop } X \{ Q \}) := \begin{cases} \mu(Q) + 1 & \text{if } Q \text{ is a sequence with a top circle} \\ \mu(Q) & \text{else} \end{cases}$$
where \( Q \equiv Q_1; \ldots; Q_l \) has a top circle if there exists a component \( Q_i \) with \( \mu(Q_i) = \mu(Q) \) such that some \( Y \) controls some \( Z \) in \( Q_i \), and \( Z \) controls \( Y \) in \( Q_1; \ldots; Q_{i-1}; Q_{i+1}; \ldots; Q_l \).

Observe that by the presence of \( \text{nil}(X) \) every primitive recursive function can already be computed by a loop program without \( \text{pred}(X) \). In fact, \( \text{pred}(X) \) can be implemented by the following program \( \text{PRED}(X) \) satisfying \( \{ X = x \}\text{PRED}(X) \{ Y = x - 1 \} \).

\[
\text{PRED}(X) := \text{nil}(Y); \text{nil}(Z); \text{loop } X [\text{nil}(Y); \text{loop } Z [\text{suc}(Y)]; \text{suc}(Z)]
\]

Thus, the loop program \( \text{loop } U [\text{PRED}(X); \text{nil}(X); \text{loop } Y [\text{suc}(X)] ] \) computes modified subtraction. But observe that this program has \( \mu \)-measure 1, and it appears that without the primitive instruction \( \text{pred}(X) \) there were no loop program of \( \mu \)-measure 0 that computes modified subtraction. On the other hand, modified subtraction is a function in \( \mathcal{E}^2 \). So to obtain the characterisation above, we treat the predecessor function as a primitive instruction – as we could do with any non-increasing function.

As with stack programs, the pivot in the proof of the characterisation theorem is to prove a “bounding theorem” stating that every function computed by a loop program of \( \mu \)-measure \( n \) can be bounded by a function in \( \mathcal{E}^{n+2} \). No doubt, to establish this result, we could follow the course as we did to prove Bounding (4.3.12) for stack programs. However, the way we choose is to benefit directly from Bounding (4.3.12).

**Lemma 4.5.3 (Stack Simulation).** Every loop program \( \mathcal{P} \) with registers \( X := x_1, \ldots, x_l \) has a stack simulation \( \mathcal{I}(\mathcal{P}) \) with stacks \( X \) (over the unary alphabet \( \{ a \} \)) such that

(a) \( \mu(\mathcal{P}) = \mu(\mathcal{I}(\mathcal{P})) \)

(b) \( \{ X = x \} \mathcal{I}(\mathcal{P}) \{ X = y \} \) if and only if \( \{ x_1 = a^{x_1}, \ldots, x_l = a^{x_l} \} \mathcal{I}(\mathcal{P}) \{ x_1 = a^{y_1}, \ldots, x_l = a^{y_l} \} \).

**Proof.** We define \( \mathcal{I}(\mathcal{P}) \) by recursion on the structure of \( \mathcal{P} \) as follows:

\[
\begin{align*}
\mathcal{I}(\text{nil}(X)) & := \text{nil}(X) \\
\mathcal{I}(\text{suc}(X)) & := \text{push}(a, X) \\
\mathcal{I}(\text{pred}(X)) & := \text{pop}(X) \\
\mathcal{I}(\mathcal{P}_1; \mathcal{P}_2) & := \mathcal{I}(\mathcal{P}_1); \mathcal{I}(\mathcal{P}_2) \\
\mathcal{I}(\text{loop } X [\mathcal{Q}]) & := \text{foreach } X [\mathcal{I}(\mathcal{Q})]
\end{align*}
\]

One can easily read off that definition that (a), (b) are true of \( \mathcal{I}(\mathcal{P}) \) for every \( \mathcal{P} \).

**Theorem 4.5.4 (Loop Bounding).** Every function \( f \) computed by a loop program \( \mathcal{P} \) of \( \mu \)-measure \( n \) has a bound \( b \in \mathcal{E}^{n+2} \), that is, \( f(\bar{x}) \leq b(\bar{x}) \) for all \( \bar{x} \).

**Proof.** Let \( \mathcal{P} \) be any loop program of \( \mu \)-measure \( n \), and let \( f \) be any \( k \)-ary function computed by \( \mathcal{P} \). Consider the corresponding function \( \mathcal{I}(f) \) computed by the stack simulation \( \mathcal{I}(\mathcal{P}) \) obtained from Lemma 4.5.3. By part (a) of that lemma \( \mathcal{I}(\mathcal{P}) \) has \( \mu \)-measure \( n \), and part (b) implies \( f(x_1, \ldots, x_k) = |\mathcal{I}(f)(a^{x_1}, \ldots, a^{x_k})| \) for all \( x_1, \ldots, x_k \). We apply Bounding (4.3.12) to obtain a length-bound \( b \in \mathcal{E}^{n+2} \) on \( \mathcal{I}(f) \), satisfying \( |\mathcal{I}(f)(\bar{a})| \leq b(|\bar{a}|) \) for all \( \bar{a} \). Now, as \( |a^{y}| = y \), we conclude that \( b \) is a bound on \( f \).

Theorem 4.5.4 implies that every function computed by a loop program of \( \mu \)-measure \( n \) belongs to \( \mathcal{E}^{n+2} \). The converse is obtained by adapting the “simulation trick” of chapter 3 to our situation. So we separate the “structure” of a program from its “growth rate”: Every \( f \in \mathcal{E}^{n+2} \) can be simulated by a loop program
\( P[f] \) of \( \mu \)-measure 0 in the sense that \( P[f] \) on \( \vec{x}, v \) outputs \( f(\vec{x}) \) whenever \( v \geq E_n^{(m)}(\max(\vec{x})) \), for some constant \( n \). As the function \( E_n^{(m)} \circ \max \) is computable by a loop program \( ME[n+1] \) of \( \mu \)-measure \( n \), the sequence \( ME[n+1] \); \( P[f] \) has \( \mu \)-measure \( n \) and computes \( f \). To succeed in that goal, we need to implement appropriately “conditional statements”.

**Definition 4.5.5 (Conditional statements).** For loop programs \( P := P_1; \ldots; P_k \) and variables \( X, Y \) we implement the conditional statement if \( X \leq Y \) then \( [P] \) as the sequence \( Q_1; Q_2; Q_3 \) defined from new variables \( U^*, V^* \) as follows:

\[
\begin{align*}
Q_1 & := \text{nil}(U^*); \text{loop } X [\text{pred}(U^*)]; \text{loop } Y [\text{pred}(U^*)] \\
Q_2 & := \text{nil}(V^*); \text{loop } U^* [\text{pred}(V^*)] \\
Q_3 & := \text{loop } V^* [P_1]; \ldots; \text{loop } V^* [P_k]
\end{align*}
\]

**Lemma 4.5.6 (Conditional).** Let \( P \) be a loop program, and let \( Z_0, \ldots, Z_n \) be the variables \( V(P) \cup \{X, Y\} \) such that \( Z_i = X \) and \( Z_j = Y \). Then \( Q := \text{if } X \leq Y \text{ then } [P] \) satisfies:

(a) If \( \{Z = \vec{z}\} P \{\vec{z} = \vec{u}\} \), then \( \{\vec{z} = \vec{z}\} Q \{\vec{z} = \vec{u}\} \) if \( z_i \leq z_j \), else \( \{\vec{z} = \vec{z}\} Q \{\vec{z} = \vec{z}\} \).

(b) \( \mu(Q) = \mu(P) \).

(c) For any \( X_0, X_1 \in \{Z_0, \ldots, Z_n\} \), \( X_0 \) controls \( X_1 \) in \( Q \) if and only if \( X_0 \) controls \( X_1 \) in \( P \).

**Proof.** Observe that \( Q_3 \) could not be not be simply \( \text{loop } V^* [P] \), because that could result in \( \mu(Q) > \mu(P) \). Hence (b), (c) are obvious. For (a), assume \( \{\vec{z} = \vec{z}\} P \{\vec{z} = \vec{u}\} \). As \( \{\vec{z} = \vec{z}\} Q_1 \{\vec{z} = \vec{z}, U^* = z_i - z_j\} \), we obtain \( \{\vec{z} = \vec{z}\} Q_1; Q_2 \{\vec{z} = \vec{z}, V^* = 1 - (z_i - z_j)\} \) and hence (a), since \( z_i \leq z_j \implies V^* = 1 \), and \( z_i > z_j \implies V^* = 0 \).

The naïve approach to compute \( f \in E^{n+2} \) by a loop program of \( \mu \)-measure \( n \) must fail, because in the case of \( f \) being defined by bounded recursion, the standard inductive construction requires to copy values computed in the recursion which then might control other recursions. Thus, these copy jobs appearing as

\[
Y = X := \text{nil}(Y); \text{loop } X [\text{succ}(Y)]
\]

can cause new top circles and result into a \( \mu \)-measure that is far beyond \( n \). To resolve that problem, in the “simulation trick” below we essentially replace “disturbing” loops (occurring in a copy job)

\[
Y := X := \text{loop } X [\text{succ}(Y)]
\]

(read increase \( Y \) by \( X \)) by the simulation (each using new variables \( U, W, H \))

\[
\text{SIM}(Y := X) := \text{nil}(U); \text{loop } V [\text{succ}(U)]; \text{if } U \leq X [\text{succ}(Y)]
\]

with if \( U \leq X [\text{succ}(Y)] := W = U; \text{loop } X [\text{pred}(W)]; \text{nil}(H); \text{succ}(H); \text{loop } W [\text{pred}(H)]; \text{loop } H [\text{succ}(Y)]
\]

Observe that if the number stored in \( V \) is greater or equal to that stored in \( X \), then \( \text{SIM}(Y := X) \) increases \( Y \) by \( X \) when executed – as \( Y := X \) does. Moreover, \( X \) does not control \( Y \) in \( \text{SIM}(Y := X) \), unlike \( Y := X \). Of course, now \( V \) controls \( Y \), and all other variables \( U, W, H \) local to \( \text{SIM}(Y := X) \). But the required simulation \( P[f] \) will be such that no other variable of \( P[f] \) controls \( V \). In that way, the problem of creating unintentional top circles is resolved. To make the construction work, however, some technical side conditions are required, too. In the following, we use the shorthand \( \text{SIM}(Y = X) \) below, and facts about it listed in (*).

\[
\text{SIM}(Y = X) := \text{nil}(Y); \text{SIM}(Y := X)
\]
\begin{align*}
& \text{(*) SIM}(Y = X) \text{ is a circle-free sequence such that } \mu(\text{SIM}(Y = X)) = 0, \text{ as the control in } \text{SIM}(Y = X) \text{ is } 
\{ U \rightarrow W, H \rightarrow Y, V \rightarrow U, V \rightarrow W, V \rightarrow H, V \rightarrow Y \}, \text{ for some local variables } U, H, W. \text{ In particular, } 
V \notin U(\text{SIM}(Y = X)) \text{ and } X, Y \notin C(\text{SIM}(Y = X)). \text{ Furthermore, if the number stored in } V \text{ is greater or equal to that stored in } X, \text{ then } \text{SIM}(Y = X) \text{ increases } Y \text{ by } X \text{ when executed.}
\end{align*}

\textbf{Lemma 4.5.7 (Loop Simulation).} Let \( V \) be any variable. For every \( f \in E^{n+2} \) one can find a simulation \( P[f] \) and a witness \( m_f \in \mathbb{N} \) such that

\begin{align*}
& (a) \quad f(\vec{x}) \leq E_{n+1}^{(m_f)}(\max(\vec{x})) \text{ for all } \vec{x}, \\
& (b) \quad \{ \vec{X} = \vec{x}, V = v \} P[f] \{ \vec{X} = \vec{x}, V = v, O = f(\vec{x}) \} \text{ for all } \vec{x}, v \text{ with } v \geq E_{n+1}^{(m_f)}(\max(\vec{x})), \\
& (c) \quad P[f] \text{ is a sequence without a (top) circle, } V \notin U(P[f]), \vec{X}, O \notin C(P[f]), \text{ and } \mu(P[f]) = 0.
\end{align*}

\textbf{Proof.} Induction on the structure of \( f \in E^{n+2} \) where \( n \geq 0 \).

Case \( f \) is an initial function. In all cases for \( f \), the claims \( (a), (b), (c) \) follow from \( (*) \) or Lemma 4.5.6.

- If \( f \) is zero then define \( P[f] := \text{nil}(O) \) and \( m_f := 0. \)
- If \( f \) is \( S \) then define \( P[f] := \text{SIM}(O = X_i) \); suc\((O)\) and \( m_f := 0. \)
- If \( f \) is \( I^{\mu}_i \) then define \( P[f] := \text{SIM}(O = X_i) \) and \( m_f := 0. \)
- If \( f \) is \( \text{max} \) then define \( P[f] := \text{SIM}(0 = X_1) \); \( f \) \( \text{if } X_1 \leq X_2 \) [\( \text{SIM}(0 = X_2) \)] \text{ and } \( m_f := 0. \)
- If \( f \) is the principal function \( E_1 \) (recall \( E_1(x) = x^2 + 2 \)), then set \( m_f := 0 \) and

\begin{align*}
P[f] &:= \text{SIM}(L = X); \text{nil}(O); \text{loop } L[\text{SIM}(O = X)]; \text{suc}(O); \text{suc}(O)
\end{align*}

for some \textit{new} variable \( L \). The case where \( f \) is \( E_{n+2} \) can be reduced to the case of bounded recursion below, since \( E_{n+2} \) can be defined by bounded recursion from \( E_{n+1} \), using \( E_{n+2} \) itself as bound.

Case \( f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_k(\vec{x})) \) with \( h, g_1, \ldots, g_k \in E^{n+2} \). The induction hypothesis yields simulations \( P[h], P[g_1], \ldots, P[g_k] \) with witnesses \( m_h, m_1, \ldots, m_k \), respectively. Thus, assuming distinct variables \( O, O_1, \ldots, O_k \), we have \( \{ \vec{X}, V = \vec{x}, v \} P[g_1] \{ \vec{X}, V = \vec{x}, v, O_i = g_i(\vec{x}) \} \) whenever \( v \geq E_{n+1}^{(m_i)}(\max(\vec{x})) \), and \( \{ \vec{O}, V = \vec{g}, v, O = h(\vec{g}) \} \) for \( v \geq E_{n+1}^{(m_h)}(\max(\vec{g})) \). Thus, by \( (*) \) and the monotonicity properties of the functions \( E_{n+1} \) (cf. Background 2.3), and assuming that the variables common to the above simulations are among \( V, \vec{X} \), the following will do:

\begin{align*}
P[f] &:= P[g_1]; \ldots; P[g_k]; P[h] \\
m_f &:= m_h + \max(m_1, \ldots, m_k)
\end{align*}

Case \( f(\vec{x}, 0) = g(\vec{x}), f(\vec{x}, y + 1) = h(\vec{x}, y, f(\vec{x})) \) and \( f(\vec{x}, y) \leq b(\vec{x}, y) \) where \( g, h, b \in E^{n+2} \). The I.H. yields simulations \( P[g], P[h] \) with witnesses \( m_g, m_h \). Thus, \( \{ \vec{X}, V = \vec{x}, v \} P[g] \{ \vec{X}, V = \vec{x}, v, O = g(\vec{x}) \} \) for \( v \geq E_{n+1}^{(m_g)}(\max(\vec{x})) \), and \( \{ \vec{X}, Y, Z, V = \vec{x}, y, z, v \} P[h] \{ \vec{X}, Y, Z, V = \vec{x}, y, z, v, O = h(\vec{x}, y, z) \} \) whenever \( v \geq E_{n+1}^{(m_h)}(\max(\vec{x}, y, z)) \). Now we choose a number \( m_b \) such that \( b(\vec{x}, y) \leq E_{n+1}^{(m_b)}(\max(\vec{x}, y)) \), and the required witness \( w_f \) is defined by \( m_f := m_b + \max(m_g, m_h) \). As for \( P[f] \), first consider the following program \( P \) which meets all requirements on \( P[f] \), except \( (c) \):

\begin{align*}
P &:= P[g]; L = Y; \text{nil}(Y); \text{loop } L[Z = O; P[h]; \text{suc}(Y)]
\end{align*}
for some new variable L. As for part (b), given any \( \vec{x}, y, v \) satisfying \( v \geq E_{n+1}^{(m_f)}(\max(\vec{x}, y)) \), observe that
\[
v \geq E_{n+1}^{(m_f)}(\max(\vec{x}, y)) \\
\geq E_{n+1}^{(m_h)}(\max(\vec{x}, i, E_{n+1}^{(m_h)}(\max(\vec{x}, i)))) \\
\geq E_{n+1}^{(m_h)}(\max(\vec{x}, i, f(\vec{x}, i))).
\]
Thus, we conclude \( \{ \vec{x}, y, v = \vec{x}, y, v \} \subseteq \{ \vec{x}, y, v = \vec{x}, y, v, O = f(\vec{x}) \} \).

Now observe that \( P \) contains the top circle \( Y \rightarrow L \rightarrow Y \). Thus, if \( P \) appeared in the body of another loop, as happens in the inductive construction of \( P[f] \) (like \( P[h] \) in \( P \)), the resulting \( \mu \)-measure were beyond 0. Furthermore, note that \( O \) controls \( Z \) in \( P \), however, by the induction hypothesis, \( Z \) has no control over any other variable in \( P[h] \). Thus, assuming that the variables common to \( P[g] \) and \( P[h] \) are among \( V, \vec{x}, O \), we obtain \( P[f] \) by modifying \( P \) to
\[
P[f] := P[g]; \text{SIM}(L = Y); \text{nil}(Y); \text{loop } L[\text{SIM}(Z = O); P[h]; \text{succ}(Y)].
\]
While (a), (b) remain valid because of (\( * \)), the I.H. and (\( * \)) ensure that (c) is now true of \( P[f] \).

**Lemma 4.5.8 (E_{n+1}-Computation).** For every \( n \geq 0 \) one can find a sequence \( E[n+1] \) with a top circle such that \( \mu(E[n+1]) = n \), and \( \{X = x\} E[n+1] \{X = E_{n+1}(x)\} \) for all inputs \( x \).

**Proof.** By induction on \( n \), where the base case for \( E_1(x) = x^2 + 2 \) is obvious. As for the step case, first recall that \( E_{n+2}(0) = 2 \) and \( E_{n+2}(x+1) = E_{n+1}(\ldots E_{n+1}(2)\ldots) \) with \( x+1 \) occurrences of \( E_{n+1} \). Using the induction hypothesis on \( n \), we therefore define
\[
E[n+2] := Y = X; \text{nil}(X); \text{succ}(X); \text{succ}(X); \text{loop } Y[E[n+1]].
\]

**Theorem 4.5.9 (Characterisation).** For every \( n \geq 0 \), a function is computable by a loop program of \( \mu \)-measure \( n \) if and only if it belongs to \( E_{n+2} \).

**Proof.** First we treat the implication “\( \Rightarrow \)”. So let \( P \) be any loop program of \( \mu \)-measure \( n \). By Loop Bounding (4.5.4) there is a function in \( E^{n+2} \) which is a bound on every function computed by a subprogram of \( P \). Thus, we can proceed by induction on the structure of \( P \) showing that every function computed by \( P \) is in \( E^{n+2} \). All cases are obvious, except possibly the case where \( P \) is of the form \( \text{loop } X[Q] \). In that case we make use of the fact that \( E^{n+2} \) is closed under bounded simultaneous recursion (cf. Background 2.2).

As for “\( \Leftarrow \)”, consider any \( f \in E^{n+2} \). Loop Simulation (4.5.7) yields a program \( P[f] \) and a witness \( m_f \) such that \( \{\vec{x} = \vec{x}, V = v\} P[f] \{Z = f(\vec{x})\} \) for \( v \geq E_{n+1}^{(m_f)}(\max(\vec{x})) \). Let \( E[n+1] \) be the program of \( \mu \)-measure \( n \) satisfying \( \{V = y\} E[n+1] \{V = E_{n+1}(y)\} \) obtained from \( E_{n+1}\text{-Computation} (4.5.8) \). Using conditional statements one defines a program \( M \) of \( \mu \)-measure 0 satisfying \( \{\vec{x} = \vec{x}\} M \{\vec{x} = \vec{x}, V = \max(\vec{x})\} \). Thus, the sequence \( M; E[n+1]; \ldots; E[n+1]; P[f] (m_f \text{ times}) \) has \( \mu \)-measure \( n \) and computes \( f \).

**Theorem 4.5.9** ensures that the measure \( \mu \) is sound with respect to computational complexity of functions computed by loop programs. This implies that \( \mu \) is also sound with respect to computational complexity of loop programs, that is, for every loop program \( P \) of \( \mu \)-measure \( n \) the function \( \text{TIME}_P \) belongs to \( E^{n+2} \), where \( \text{TIME}_P(\vec{x}) \) is the number of imperatives executed in a run of \( P \) on \( \vec{x} \). To see this, given any loop
program $P$ of $\mu$-measure $n$, let $P^\#$ result from $P$ by replacing every occurrence of an imperative $\text{imp}$ with $\text{imp; suc}(V)$, where $V$ is any fixed new variable. Then $\text{TIME}(P) \equiv \text{nil}(V) ; P^\#$ has $\mu$-measure $n$ and computes $\text{TIME}_P$, that is, $\{\vec{X} = \vec{x}\} \text{TIME}(P) \{V = \text{TIME}_P(\vec{x})\}$. Hence $\text{TIME}_P \in \mathcal{E}^{n+2}$ by Theorem 4.5.9.

4.6. Sound, adequate and complete measures

In this chapter a purely syntactical method for analysing the impact of nesting loops in stack programs on computational complexity has been presented. In particular, the method separates programs which run in polynomial time from programs running in exponential time. More generally, the method separates uniformly programs with running time in $\mathcal{E}^{n+2}$ from programs with running time in $\mathcal{E}^{n+3}$.

One might ask how successful this project can be, that is e.g., does every stack program with polynomial running time receive $\mu$-measure 0? In this section we will shed some light upon the limitations of any such method, however, bring out that the results we have achieved are about as good as one can hope for.

Definition 4.6.1. Assume an arbitrary imperative programming language $L$ and an arbitrary program $P$ in $L$. $P$ is feasible if every function computed by $P$ is in $\mathcal{F}P$. $P$ is honestly feasible if every subprogram of $P$ is feasible. $P$ is dishonestly feasible, or dishonest for short, if $P$ is feasible, but not honestly feasible.

Note that if a function is computable by a feasible program, then it is also computable by an honestly feasible program.

For honestly feasible programs, every subprogram can be simulated by a Turing machine running in polynomial time. Dishonest programs fall into two groups: One consists of those programs which only compute functions in $\mathcal{F}P$, but with super-polynomial running time, the other consists of programs which run in polynomial time, but some subprograms have super-polynomial running time if executed separately. Typical of the latter group are programs of the form $R; \text{if}<\text{test}>[Q]$ where $R$ is a program which runs in polynomial time, $<\text{test}>$ is a test that always fails, and $Q$ is an arbitrary program with super-polynomial running time. Another example is a program of the form $P; Q$ where $Q$ runs in time $O(2^x)$, but where $P$ is an honestly feasible program which assures that $Q$ always is executed on ”logarithmically large input”.

Obviously, we cannot expect to separate (by purely syntactical means) the feasible programs from the non-feasible ones if we take into account dishonest programs. Thus, it seems reasonable to restrict our discussion to the honestly feasible programs, and after all, it is the computational complexity inherent in the code we want to analyse and recognise. But even then, our project is bound to fail.

Definition 4.6.2. Given any stack programming language $L$, a measure on $L$ is a computable function $\nu: L$-programs $\rightarrow \mathbb{N}$.

Definition 4.6.3. Let $L$ be an arbitrary (reasonable) stack programming language containing the core language defined in section 5, and let $\nu$ be a measure on $L$. The pair $(\nu, L)$ is called

- sound if every $L$-program of $\nu$-measure 0 is feasible,
- complete if every honestly feasible $L$-program has $\nu$-measure 0, and
- adequate if every function in $\mathcal{F}P$ is $L$-computable with $\nu$-measure 0.

As seen above, core programs are the backbones of more general stack programs and they comprise those stack manipulations which do contribute to computational complexity. Let $C$ denote the set of core programs defined in section 5, and let $\mu$ be the measure on core programs as defined in section 4. The next theorem is good news.
Theorem 4.6.4. The pair \((\mu, C)\) is sound and complete.

Proof. Soundness follows directly from Corollary 4.4.5. As for completeness, assume that \(P\) were an honestly feasible core program with \(\mu(P) > 0\). Then \(P\) contained a loop \(\text{foreach } X [Q]\) where \(Q\) is a sequence \(Q_1; \ldots ; Q_l\) with a top circle. So \(Q\) contained a component \(Q_i\) such that some \(Y\) controls some \(Z\) in \(Q_i\), and \(Z\) controls \(Y\) in \(Q_1; \ldots ; Q_{l-1}; Q_{l+1}; \ldots ; Q_l\). Now observe that if \(U\) controls \(V\) in a core program \(R\), then by monotonicity each time \(R\) is executed the size of stack \(V\) increases at least by the size of stack \(U\). So we conclude that each time \(Q\) is executed the size of at least one stack in \(Q\) were doubled. Thus, \(P\) contained a non-feasible subprogram, contradicting the assumption that \(P\) is honestly feasible.

The pair \((\mu, C)\) is obviously not adequate. As core programs are length-monotonic, there are plenty of functions in \(\text{FP}\) which are not \(C\)-computable, let alone \(C\)-computable with \(\mu\)-measure 0. However, wouldn’t it be nice if we could extend \((\mu, C)\) to an adequate pair and still preserve both soundness and completeness? Well, it is not possible.

Theorem 4.6.5. Let \((\nu, L)\) be a sound and adequate pair. Then \((\nu, L)\) is incomplete, that is, there exists an honestly feasible program \(P \in L\) such that \(\nu(P) > 0\).

Proof. Let \(\{P_i\}_{i \in \omega}\) be an effective enumeration of the Turing machines with alphabet \(\{0, 1\}\). Let \(n\) be a fixed natural number. It is well-known that there is a function \(f_n \in \text{FP}\) satisfying

\[
f_n(x) = \begin{cases} 
1 & \text{if } \rho_n \text{ (on the empty input) halts within } |x| \text{ steps} \\
0 & \text{else}
\end{cases}
\]

Also well-known is that it is undecidable whether \(\rho_i\) halts. Since \((\nu, L)\) is adequate and sound, there is an honestly feasible program \(Q \in L\) of \(\nu\)-measure 0 such that

\[\{Y = y\} \subset Q \{\text{if } \rho_n \text{ does not halt within } |y| \text{ steps then } Z = \varepsilon \text{ else } Z = 1\}\]

Moreover, such a program \(Q\) can be effectively constructed from \(n\), that is, there exists an algorithm for constructing \(Q\) from \(n\). Since \(L\) contains the core language, the program

\[P := \text{foreach } X [Q]; \text{foreach } Z [\text{foreach } V [\text{push}(1, W)]; \text{foreach } W [\text{push}(1, V)]]\]

is also in \(L\), where \(X, V, W\) are new stacks. Now, if \(\rho_n\) never halts then \(\text{foreach } V [\text{push}(1, W)]\) will never be executed, whatever the inputs to \(P\). Thus, if \(\rho_n\) never halts, then \(P\) is honestly feasible. In contrast, if \(\rho_n\) halts after \(s\) steps, say, then part \(\text{foreach } V [\text{push}(1, W)]\) and part \(\text{foreach } W [\text{push}(1, V)]\) will be executed each time the body of the outermost loop is executed whenever \(Y = y\) with \(|y| \geq s\). Each such execution implies that the height of stack \(V\) is at least doubled. Thus, if \(\rho_n\) eventually halts, then \(P\) is not feasible. In other words, \(P\) is honestly feasible if and only if \(\rho_n\) never halts. As \(P\) is effectively constructible from \(n\), we conclude that \((\nu, L)\) cannot be complete. For if \((\nu, L)\) were complete, then \(\rho_n\) would never halt if and only if \(\nu(P) = 0\). This would yield an algorithm which decides whether \(\rho_n\) halts: Construct \(P\) from \(n\) and then check whether \(\nu(P) > 0\). Such an algorithm does not exist.

Notably, Theorem 4.6.5 relates to Gödel’s First Incompleteness Theorem. The latter implies that if a first order language is expressive enough, then there is no algorithm which can identify the true statements of the language. Theorem 4.6.5 says that when a programming language is sufficiently expressive, then there is no algorithm which can identify the honestly feasible programs of the language.
5. Characterising Polynomial-Time by Higher Type Recursion

In this chapter it is shown how to restrict recursion on notation in all finite types so as to characterise the polynomial-time computable functions. The restrictions are obtained by using a ramified type structure, and by adding linear concepts to the lambda calculus. This gives rise to a system RA of ramified affinable terms which allows one top recursion in any finite type, and any number of side recursions in all finite types.

It is shown that for each closed RA term $t$ of type level 1, one can find a polynomial $p_t$ such that for all numerals $\bar{n}$, one can compute the normal form $nf(t\bar{n})$ in time $p_t(|\bar{n}|)$. Thus, $t$ denotes a polynomial-time computable function. The converse also holds, as each polynomial-time computable function is computed by some RA term. Observe that there are two normalisations required to compute the normal form $nf(t\bar{n})$: One is to normalise $t\bar{x}$ to $u$, say, which may take a long time (super-polynomial in the length of $t$), the other is to normalise $u[\bar{x}/\bar{y}]$, which will take polynomial time in the length of $\bar{n}$. One may view the first normalisation as a (complex) compilation step, producing efficient code.

Recall the central ideas outlined in the Introduction. One problem with higher-type recursion on notation (cf. Background) is that non-primitive recursive growth rate is obtained by applying the computed functional $\lambda u. V v$. Another problem is that non-primitive recursive growth rate is obtained by using a ramified type structure, and any number of side recursions in all finite types.

Thus, to set up a subsystem RA of G"odel's $T$ in which the use of higher-type recursion on notation is tamed and controlled in a purely syntactic fashion, we introduce at the same time both ramification on the type structure and linearity conditions for the use of computed functionals in recursions.

In terms of ramification, some objects are labelled with a mark $!$, that is, we enrich the type structure with the formation of types $!\sigma$, called complete types; all other types are called incomplete. Intuitively,

objects of \begin{align*}
\text{complete higher type} & \quad \text{can be used in any non-linear way} \\
\text{incomplete higher type} & \quad \text{can be used in a certain linear way only} \\
\text{type} & \quad \text{control at most one top recursion} \\
\text{type} & \quad \text{control no recursion.}
\end{align*}

By ramification one can rule out non-primitive recursive growth rate, and one can generalise the concept that computed values in a recursion must not control other recursions by requiring that:

(i) Recursion in type $\sigma$ is only admitted in the form $\text{RN}_{\sigma}^t h n$ for $!$-free types $\sigma$, called safe types, and for $h, n$ of complete types $!(\imath \to \imath \to \imath)$, $!\imath$ respectively.

(ii) Terms of complete types must not contain incomplete free variables.

Observe that (i),(ii) do not rule out the nesting of computed functionals in recursions, such as in the definition of the function $d$ above. This is precluded by another concept, called affinability, which is central to the system and expresses the linearity constraints for bound variables of incomplete higher type:
(iii) The formation of terms $\lambda x^\sigma . r$ for an incomplete higher type $\sigma$ is admitted only if $x^\sigma$ is affinable in $r$, that is, either $r$ has at most one free occurrence of $x^\sigma$, or else $r$ has a subterm $a$ of type $\iota$ with a single free occurrence of $x^\sigma$ such that every free occurrence of $x^\sigma$ in $r$ is in an occurrence of $a$, which is to say, $r$ is of the form

$$r = \ldots a \ldots a \ldots a \ldots$$

such that the free occurrences of $x^\sigma$ in $r$ are separated by one and the same ground type context $a$.

These restrictions permit one top recursion in any type, and as many side recursions in any type as desired. In fact, RA programs compute exactly the polynomial-time computable functions.

5.1. Marked types and terms

In this section system $T$ based on recursion on notation (cf. Background) is extended with the formation of marked types $!\sigma$ and terms of these types, thereby providing basic definitions and concepts relevant for the definition of or proofs of statements about system RA.

Definition 5.1.1 (Types and level).

- **Types** are $\iota$, and inductively, if $\sigma$, $\tau$ are types, then so are $\sigma \to \tau$ and $!\sigma$.
- The **level** $l(\sigma)$ of $\sigma$ is inductively defined by $l(\iota) := 0$, $l(!\sigma) := l(\sigma)$, $l(\sigma \to \tau) := \max\{l(\tau), 1 + l(\sigma)\}$.
- **Ground** types are the types of level 0, and **higher** type is any type of level at least 1.

For example, $!!\iota$ is a ground type, but $\iota \to \iota$ is a higher type. We assume that $!$ binds tighter than $\to$, and that $\to$ associates to the right. Thus, a type of the form $\sigma \to !\tau \to \rho$ reads $\sigma \to (!\tau \to \rho)$. One writes $\vec{\sigma} \to \tau$ for $\sigma_1 \to \cdots \to \sigma_n \to \tau$. If $\vec{\sigma}$ is empty, then $\vec{\sigma} \to \tau$ is just $\tau$.

To ensure (i) above for RA terms, we generalise “complete” and “incomplete” as follows:

Definition 5.1.2 (Complete and incomplete types). Types fall into two groups:

- **Complete** types of the form $\vec{\sigma} \to !\tau$
- **Incomplete** types of the form $\vec{\sigma} \to \iota$.

**Safe** types are $!$-free types. Thus, ground types are either safe or complete.

The **constant symbols** are listed below, with their types.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\iota$</td>
<td></td>
</tr>
<tr>
<td>$s_0$</td>
<td>$\iota \to \iota$</td>
<td>(binary successor $x \mapsto 2 \cdot x$)</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$\iota \to \iota$</td>
<td>(binary successor $x \mapsto 2 \cdot x + 1$)</td>
</tr>
<tr>
<td>$p$</td>
<td>$\iota \to \iota$</td>
<td>(binary predecessor $x \mapsto \lfloor \frac{x}{2} \rfloor$)</td>
</tr>
<tr>
<td>$c_\sigma$</td>
<td>$\iota \to \sigma \to \sigma \to \sigma \to \sigma$</td>
<td>for $\sigma$ safe (cases in type $\sigma$)</td>
</tr>
</tbody>
</table>

**Terms** are built from these constants and typed variables $x^\sigma$ by introduction and elimination rules for the two type forms $\sigma \to \tau$ and $!\sigma$, and by formation of recursion terms\(^1\), that is,

$$\begin{align*}
(\lambda x^\sigma . r)^{\sigma \to \tau}, \\
(r^{\sigma \to \tau} \cdot s^\sigma)^{\tau}, \\
(1_{\sigma^\sigma}^{\sigma})^\sigma, \\
(\kappa r^{1_{\sigma^\sigma}})^{\sigma}, \\
(RN_{\rho} g^\rho h_{l(1_{\iota \to \rho} \to \rho)} n^{l_{\iota}})^\rho 
\end{align*}$$

\(^1\)RA programs will be evaluated using a kind of sharing concept, defined inside RA. To equip the resulting subclass SRA with nice properties, it is convenient to treat $RN_\sigma$ as a term former rather than as a constant.
Recall that a binary numeral is either 0 or it is of the form $s_{i_1} \ldots s_{i_k} 0$ with $i_j \in \{0, 1\}$ for $j = 1, \ldots, k$, $k \geq 0$. Binary numerals distinct from 0 are denoted by $s_i$. The conversion rules are then as expected:

\[
\begin{align*}
\lambda\! x\! .\! r\! s & \mapsto r[s/x] & c_\sigma 0 & \mapsto r \\
\kappa(!r) & \mapsto r & c_\sigma (s_i n) & \mapsto t_i \\
s_0 0 & \mapsto 0 & \text{RN} & \mapsto g \\
p_0 0 & \mapsto \text{RN} & \kappa h! (s_i n) & \mapsto \kappa h! (s_i n) (\text{RN} g h! n)
\end{align*}
\]

The length $|t|$ of a term $t$ is defined by $|x| = |c| = 1$; $|\lambda x. r| = |!r| = |r| = |r\kappa| = |r| + 1$; $|rs| = |r| + |s| + 1$.

Redexes are subterms shown on the left-hand side of conversion rules above. A term is in normal form if it does not contain a redex. We write $t \rightarrow t'$ if $t'$ results from $t$ by converting one redex; we say $t$ reduces to $t'$, or $t'$ is a reduct of $t$, if $t \rightarrow^* t'$ where $\rightarrow^*$ denotes as usual the reflexive and transitive closure of $\rightarrow$. Observe that $\rightarrow$ is consistent with substitution (cf. Background).

Using standard techniques (see e.g. [62, 29] for proofs of normalisation in Gödel’s system $T$), one also obtains strong normalisation for this version of system $T$: Every maximal reduction sequence starting with a term $t$ is finite and results in a unique term $\text{nf}(t)$ in normal form.

One writes $\text{FV}(t)$ for the set of free variables of $t$, and $\text{FO}(x, t)$ for the number of free occurrences of $x$ in $t$. Say that a term is complete, incomplete, safe, or ground if its type is.

Now, as with system $T$, types and terms are interpreted over the set-theoretical function spaces. In other words, in the semantics we identify objects of type $! \sigma$ with those of type $\sigma$, since we are only interested in the computational behaviour of terms. Accordingly, $H_{t! \sigma} := H_\sigma$, and the value $[t]_\varphi$ of a term $t$ in an environment $\varphi$ is defined as usual, except that $[! r]_\varphi := [r]_\varphi$ and $[\kappa r]_\varphi := [r]_\varphi$.

### 5.2. RA terms

In this section we will define system RA, the design of which is motivated by the desire to have a subclass of terms which is closed under reduction, and for which it is decidable whether a given term belongs to RA.

Two subterms $a_i$ and $a_j$ occurring in a term $t$ are scope equivalent if whenever $\lambda y$ binds a variable free in either $a_i$ or $a_j$, then both $a_i$ and $a_j$ lie within the scope of the $\lambda y$.

**Definition 5.2.1 (Affinability).** Let $x$ be an incomplete variable, and let $r$ be a term.

- An affination of $x$ in $r$ is a ground type subterm $a^{x}$ with $\text{FO}(x, a) = 1$ such that every free occurrence of $x$ in $r$ is in an occurrence of $a$ in $r$, and the occurrences of $a$ in $r$ are scope equivalent in $r$.

- $x$ is affinable in $r$ if there is an affination of $x$ in $r$ or $\text{FO}(x, r) \leq 1$.

**Definition 5.2.2 (System RA).** Ramified affinable terms, or RA terms for short, are built like terms except that introduction of $!$, application and lambda abstraction are modified as follows:

\[(! I)\] If $r^{\sigma}$ is an RA term with $\text{FV}(r)$ all complete, then $(! r)^{\sigma}$ is an RA term.

\[(A)\] If $r^{\sigma \rightarrow r}$ and $s^{\sigma}$ are RA terms, then $(r s)^{\sigma}$ is an RA term; provided that if $r$ is complete, then $\text{FV}(s)$ are all complete.

\[(L)\] If $r^{\tau}$ is an RA term, then $(\lambda x^{\sigma} r)^{\sigma \rightarrow}^{\tau}$ is an RA term; provided that if $\sigma$ is incomplete, then $x$ is affinable in a reduct of $r$. 

Observe that every variable of type $\iota$ is trivially affinable in every term, because it is an affination of itself. Thus the restriction on lambda abstracted variables $x^\sigma$ in (L) applies only to higher-type incomplete variables. Furthermore, observe that subterms of RA terms are in RA, too.

Polynomially-growing functions, such as string concatenation $\oplus$ and string multiplication $\otimes$, are easily definable in RA:

$$
t_{\oplus} := \lambda x^t y^t. \text{RN}_{t} (x !(\lambda u^t v^t. c (\kappa u^t) 0 (s_0 v^t) (s_1 v^t)) y) \\
t_{\otimes} := \lambda x^t y^t. \text{RN}_{t} 0 !(\lambda u^t v^t. t_{\oplus} v x) y
$$

In system RA computed functionals in higher-type recursions can only be passed to safe affinable input positions. Admitting recursion $\text{RN}_{\sigma}$ for incomplete $\sigma$ would result in exponential growth. For example, writing the equations $d'(0, n) = n$ and $d'(s_1(n), n) = d'(m, d'(m, \text{sq}(n)))$ for $s_1(n) \neq 0$, with $\text{sq}(n) = 2^{2|n|}$, as a recursion in type $t_1 \rightarrow t$, one could define the function $d'$ satisfying $d'(m, n) \geq n^{2|n|}$. To see this, define $\text{sq}$ by $t_{\text{sq}} := \lambda y^{t_1}. \text{RN}_{t} (s_1 0) !(\lambda u^{t_1} v^{t_1}. s_0 (s_0 v^{t_1})) y$, and then $d'$ by

$$
t_{d'} := \lambda x^{t_1}. \text{RN}_{t_1 \rightarrow t} (\lambda y^{t_1}. \kappa y) !(\lambda u^{t_1} V^{t_1 \rightarrow t} y^{t_1}. V !((t_{\text{sq}} y) x).
$$

Affinability is designed to rule out nested occurrences of computed functionals in recursions, such as that used to define $d$ above. It requires that if we lambda abstract a higher-type incomplete variable $x$ in $r$, then either $\text{FO}(x, r) \leq 1$ or else the free occurrences of $x$ in $r$ can be separated by the occurrences of one and the same ground type context $a$, the affination of $x$ in $r$. But observe that if $x$ is affinable in $r$ and $r \rightarrow r'$, then $x$ need not be affinable in $r'$. For example, suppose that $x$ has an affination $a$ in $r$ such that $\text{FO}(x, r) \geq 2$, and $r'$ is obtained from $r$ by reducing a single occurrence of $a$ to some $a'$ such that $\text{FO}(x, a') = 1$ and $a'$ is no subterm of $a$, (for example, $a = c_{\sigma}(s_1 n) r t_0 t_1 r$ with $x \in \text{FV}(t_i)$ and $a' = t_i r$), say

$$r = \ldots a a a a \ldots \rightarrow \ldots a a a a \ldots = r'.
$$

Then neither $\text{FO}(x, r') \leq 1$ nor does $x$ have an affination in $r'$. Thus, to obtain a system that is closed under reduction, modified lambda abstraction (L) requires that $x$ is affinable in a reduct of $r$, the idea being that every reduction locally performed on an occurrence of $a$ in $r$ can be successively performed on all occurrences of $a$ in $r$, thus leading to a reduct $r'' = \ldots a' a' \ldots a' \ldots$ of $r$ in which $a'$ is an affination of $x$. In fact, closure under reduction is obtained by showing (cf. Affinability 5.3.3) that if $x$ is affinable in an RA term $r$ and if $r \rightarrow r'$, then $x$ is affinable in a reduct of $r'$.

It remains to comment on scope equivalence required for the occurrences of an affination of a higher-type incomplete bound variable (5.2.1). This requirement simply rules out the hiding of nested occurrences of computed functionals in recursions. To see this, consider the following modified definition of $d$ above:

$$
\lambda x^{t_1}. \text{RN}_{t_1 \rightarrow t} s_1 (\lambda u^{t_1} V^{t_1 \rightarrow t} y^t. (\lambda y^t. V y)(V y)) x
$$

Without scope equivalence, the two occurrences of $V y$ would give an affination of $V$ in that term, resulting in functions of exponential growth. Of course, such hiding of nested occurrences of computed functionals in recursions, could be ruled out as well by requiring that every name for a bound variable may be used only once. But this would seriously interfere with readability, and moreover, it would impose an unnecessary constant need for bounded renaming when evaluating RA programs.

5.3. Closure of RA under reduction

In this section we prove that system RA is closed under reduction. To this end, we first establish some other properties of RA which confirm some of the intuition expressed above.
Lemma 5.3.1 (Completeness). Every complete RA term \( r \) has complete free variables only.

Proof. We proceed by induction on the structure of complete RA terms \( r \). For type reasons, \( r \) is neither a constant nor a recursion. If \( r \) is a variable, then we are done. If \( r \) is \(!s\), then \( \text{FV}(r) \) all are complete by (!I). If \( r \) is \( \kappa s \), then \( s \) has type \( !\sigma \), hence \( \text{FV}(r) \) all are complete by the induction hypothesis on \( s \). If \( r \) is \( s^{\sigma \rightarrow t^\sigma} \), then \( s \) is complete by the hypothesis on \( r \), and by (A) \( t \) contains complete free variables only. Hence the claim for \( r \) follows from the induction hypothesis on \( s \). If \( r \) is \( \lambda x.s \), then \( s \) is complete, and \( \text{FV}(r) \) all are complete by the induction hypothesis on \( s \).

\( \square \)

Lemma 5.3.2 (Closure under Substitution). If \( r, s \) are RA terms, then so is \( r[s/x] \).

Proof. We proceed by induction on the structure of \( r \). If \( r \) is a variable or a constant, then \( r[s/x] \) is either some \( s_i \), or else it is \( r \). In either case, \( r[s/x] \) is an RA term.

Case \( r \) is \(!s\). Then \( r[s/x] = !s[s/x] \) and by (!I) \( \text{FV}(s) \) all are complete. Now \( s[s/x] \) is an RA term by the induction hypothesis on \( s \), and by Completeness (5.3.1) each \( s_i \) being substituted for a free occurrence of \( x_i \) in \( s \) has complete free variables only. Hence by (!I) \( !s[s/x] \) is in RA.

Case \( r \) is \( \kappa s \). Then \( r[s/x] = \kappa s[s/x] \), so the claim follows from the induction hypothesis on \( s \).

Case \( r \) is \( s^{\sigma \rightarrow t^\sigma} \). Then \( r[s/x] = (s[s/x])(t[s/x]) \), and both \( s[s/x] \) and \( t[s/x] \) are RA terms by the induction hypothesis. We may assume that \( s \) is complete, hence by (A) \( \text{FV}(t) \) all are complete. Then every \( s_i \), substituted for a free occurrence of \( x_i \) in \( t \) is complete. Thus Completeness (5.3.1) implies that \( t[s/x] \) has complete free variables only. Now (A) gives that \( r[s/x] \) is in RA.

Case \( r \) is \( \lambda x.s \). We may assume that \( x \notin \bar{x} \), and \( x \notin \text{FV}(s_i) \) for all \( s_i \). Hence \( r[s/x] = \lambda x.s[s/x] \), and \( s[s/x] \) is an RA term by the induction hypothesis. We may assume that \( x \) is incomplete, so \( x \) is affinable in a reduct \( r'' \) of \( r' \). Hence \( r[s/x] \) is affinable in the reduct \( r''[s/x] \) of \( r'[s/x] \), showing that \( r[s/x] \) is in RA.

Case \( t \) is a recursion \( RN_{\sigma,ghn} \). We conclude \( t \in RA \) from the induction hypothesis on \( g, h, n \).

\( \square \)

Lemma 5.3.3 (Affinability). Let \( r \) be an RA term such that \( x \) is affinable in \( r \). If \( r \rightarrow r' \), then \( x \) is affinable in a reduct of \( r' \).

Proof. Induction on the structure of \( r \), assuming \( r \rightarrow r' \) and that \( x \) is affinable in \( r \), and \( \text{FO}(x,r) \geq 1 \).

Case \( \text{FO}(x,r) = 1 \). We may assume \( \text{FO}(x,r') \geq 2 \). By Completeness (5.3.1) the only conversion rule capable of multiplying an incomplete variable is a \( \beta \) conversion. Thus, \( r' \) results from \( r \) by converting a redex \( (\lambda y.s)t \) where the unique free occurrence of \( x \) in \( r \) is in \( t \). As \( x \) is incomplete, Completeness (5.3.1) implies that \( t \) is incomplete, and so is \( y \). Hence \( y \) is affinable in a reduct \( s' \) of \( s \), and we may assume that \( y \) has an affination \( b \) in \( s' \). As the unique free occurrence of \( x \) in \( r \) is in \( t \), we obtain that \( b[t/y] \) is an affination of \( x \) in the reduct \( s'[t/y] \) of \( s[t/y] \), showing that \( x \) is affinable in a reduct of \( r' \).

Case \( \text{FO}(x,r) \geq 2 \). Then \( x \) has an affination \( a \) in \( r \), as \( x \) is affinable in \( r \). Let \( r' \) result from \( r \) by converting a redex \( R \) in \( r \). If \( R, a \) are separated in \( r \), then \( a \) is an affination of \( x \) in \( r' \). Otherwise either \( R \) occurs in \( a \) or \( a \) is a proper subterm of \( R \).

Subcase \( R \) occurs in \( a \). Hence \( r' \) results from \( r \) by reducing an occurrence of \( a \) to some \( a' \). As \( \text{FO}(x,r) \geq 2 \), \( a \) is a proper subterm of \( r \). Thus, as \( x \) is affinable in \( a \), the induction hypothesis on \( a \) yields that \( x \) is affinable in a reduct \( a'' \) of \( a' \). We conclude that \( x \) is affinable in the reduct \( r'' \) of \( r' \) obtained from \( r \) by successively reducing all occurrences of \( a \) in \( r \) to \( a'' \).

Subcase \( a \) is a proper subterm of \( R \). We may assume \( \text{FO}(x,r') \geq 2 \). Inspecting all possible forms of \( R \) other than a \( \beta \) redex, we see by Completeness (5.3.1) that in either case, \( a \) is still an affination of \( x \) in
So assume that $R$ is $(\lambda y.s)t$. First consider the case where $a$ occurs in $t$. Now observe that $a$ may occur in $s$, but in that case, scope equivalence implies $y \notin \text{FV}(a)$. Thus, if $a$ occurs in $t$, we conclude from $\text{FO}(x, r') \geq 2$ that $a$ is still an affination of $x$ in $r'$ – where the occurrences of $a$ in $r$ are multiplied in $r'$ in case $\text{FO}(y, s) \geq 2$. It remains to consider the case where $a$ occurs in $s$. If $y \in \text{FV}(a)$, then by scope equivalence all occurrences of $a$ in $r$ must be in $s$, hence $a[t/y]$ is an affination of $x$ in $r'$. Otherwise if $y \notin \text{FV}(a)$, then $a$ is still an affination of $x$ in $r'$.

\[\square\]

**Theorem 5.3.4 (Closure under Reduction).** If $r \rightarrow r'$ and $r$ is in RA, then so is $r'$.

**Proof.** We proceed by induction on the height $h(r)$ of the reduction tree for $r$, and side induction on the structure of $r$, assuming $r \rightarrow r'$ in each case for $r$. The cases where $r$ is a variable or a constant are ruled out, since these terms are not reducible.

**Case $r$ is $!s$.** Then $\text{FV}(s)$ all are complete, and $r'$ is $!s'$ such that $s \rightarrow s'$. The side induction hypothesis on $s$ yields that $s'$ is in RA. Hence $r'$ is in RA, as $\text{FV}(s') \subseteq \text{FV}(s)$.

**Case $r$ is $\kappa s$.** Then either $s$ is $!s'$ and $r'$ is $s'$, in which case we are done, or else $r'$ is $\kappa s'$ and $s \rightarrow s'$. In the latter case, the side induction hypothesis on $s$ implies that $r'$ is an RA term.

**Case $r$ is $s(a)$ in RA.** If $r'$ results from $r$ by reducing either $s$ or $t$, then $r'$ is in RA by the side induction hypothesis on the reduced subterm. Otherwise $st$ is itself a redex, and $r'$ results from $r$ by converting that redex. So it suffices to show that if the left-hand side term of a conversion rule is in RA, then so is the right-hand side term. This is obvious for $s_0$, $p$, $\kappa$ or $c$ conversions, and for $\beta$ conversions we apply Substitution (5.3.2).

In case of an $\text{RN}$ conversion, it suffices to consider the rule $\text{RN}_{\sigma}gh!(s_1n) \rightarrow \kappa h!(s_1n)(\text{RN}_{\sigma}gh!n)$. Observe that $\kappa h$ is an incomplete RA term. Hence by (A) the right-hand side term is in RA.

**Case $r$ is $\lambda x.s$.** Hence $r'$ is $\lambda x.s'$ and $s \rightarrow s'$. We may assume that is $x$ incomplete. Hence $x$ is affinable in a reduct $s^*$ of $s$. We have to show that $x$ is affinable in a reduct of $s'$. If $s^*$ is $s$, then the claim follows directly from Affinability (5.3.3). Otherwise we find ourselves in the situation:

$$s \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \equiv s^*, \ n \geq 1, \text{ and } x \text{ is affinable in } s^*$$

Now the side induction hypothesis on $s$ yields that $s'$, $s_1$ are in RA. Successively applying the main induction hypothesis on $s_1, \ldots, s_{n-1}$, we obtain that $s^*$ is in RA, and so does every reduct $s^{**}$ of $s^*$, for $h(s^{**}) \leq h(s^*) < h(r)$. As $x$ is affinable in $s^*$, we conclude from Affinability (5.3.3) that $x$ is affinable in the reduct $\text{nf}(s^*) = \text{nf}(s) = \text{nf}(s')$ of $s'$. To see this, we show that if $t$ is in RA such that every reduct of $t$ is in RA, and if $y$ is affinable in $t$, then $y$ is affinable in $\text{nf}(t)$. For the proof, we proceed by induction on the height of the reduction tree for $t$. Suppose that $t \rightarrow t'$ for some $t'$. By Affinability (5.3.3) $y$ is affinable in a reduct $t''$ of $t'$. Since $t''$ and all reducts of $t''$ are in RA, the induction hypothesis on $t''$ gives that $y$ is affinable in $\text{nf}(t'') = \text{nf}(t)$.

**Case $r$ is $\text{RN}_{\sigma}gh!n$.** Then either $r'$ results from $r$ by converting that outermost $\text{RN}$ redex, or else $r'$ results from $r$ by reducing a component of $r$. The former case has been treated above, in the latter case $r' \in RA$ follows from the induction hypothesis on the reduced component.

\[\square\]

**Note.** Strong normalisation for terms and closure of RA under reduction imply that every reduction sequence for an RA term $r$ terminates in a unique normal RA term $\text{nf}(r)$. In particular, if $r$ is closed and ground, then $\text{nf}(r)$ is a numeral denoting $[r]$, that is, a binary numeral preceded by any number of $!$'s.
5.4. Analysis of Normal Terms

In this section we analyse the structure of normal terms, exploring the effect of the type restrictions with respect to those terms defining number-theoretical functions. By Type Analysis (5.4.3) those terms use safe or ground type variables only.

Lemma 5.4.1 (Kappa). If \( \kappa \tau \) is a normal term, then \( \tau \) is of the form \( \kappa(\kappa \ldots \kappa(X\vec{r}_1)\vec{r}_2 \ldots)\vec{r}_n \) for some complete variable \( X \).

Proof. Induction on the structure of normal terms \( \tau \). For type reasons, \( \tau \) is of the form \( U\vec{s} \) where \( U \) is neither a constant nor a symbol \( \text{RN}_\sigma \). Since \( \kappa \tau \) is normal, \( U \) is not the symbol \( ! \). If \( U \) is a variable \( X \), then it has a type \( \vec{s} \rightarrow !\tau \) and we are done. If \( U = \kappa \), then \( \#\vec{s} \geq 1 \) and \( s_1 \) has type \( !\sigma \). Thus, \( s_1 \) has the required form by the induction hypothesis on \( s_1 \), and so does \( \kappa\vec{s} = \tau \).

Definition 5.4.2. For types \( \sigma, \rho \) we say that \( \sigma \) occurs strictly left in \( \rho \), denoted \( \sigma \prec \rho \), if either \( \rho = \rho_1 \rightarrow \rho_2 \) and \( (\sigma \preceq \rho_1 \text{ or } \sigma \prec \rho_2) \), or else \( \rho = !\rho' \) and \( \sigma \prec \rho' \). As usual \( \preceq \) stands for \( \lhd \) or \( = \).

Lemma 5.4.3 (Type Analysis). Let \( t^\sigma \) be a normal term, and let \( x^\tau \) be a higher-type non-safe variable of \( t \). Then \( (\ast) \) \( \tau \prec \sigma \) or \( \tau \preceq \rho \) for some \( y^\rho \in \text{FV}(t) \).

Proof. We proceed by induction on the structure of normal terms \( t \) of any type \( \sigma \).

Case \( t = \lambda x^\sigma . r^\sigma \). If \( x = z \), then \( \tau = \sigma_1 \prec (\sigma_1 \rightarrow \sigma_2) = \sigma \). Otherwise the induction hypothesis on \( r \) yields either \( \tau \prec \sigma_2 \), implying \( \tau \prec (\sigma_1 \rightarrow \sigma_2) = \sigma \), or else \( \tau \preceq \rho \) for some \( y^\rho \in \text{FV}(r) \). In the latter case, if \( y \neq z \) then \( y \in \text{FV}(t) \), and if \( y = z \), then \( \tau \preceq \sigma_1 \prec \sigma \). In either case, \( (\ast) \) is true of \( t \).

Case \( t = X\vec{r} \). If \( \#\vec{r} = 0 \) or \( X = x \), then the second alternative of \( (\ast) \) holds. Otherwise \( x \) occurs in some \( r_i^\sigma \), and so the induction hypothesis yields \( \tau \prec \sigma_i \), implying \( \tau \prec \text{type}(X) \), or else \( \tau \preceq \rho \) for some \( y^\rho \in \text{FV}(r_i) \subseteq \text{FV}(t) \). In either case, the second alternative of \( (\ast) \) holds.

Case \( t = c\vec{r} \) where \( c \) is either a constant or \( \text{RN}_\sigma \). Then either \( x \) occurs in some safe or ground type subterm, or else \( c = \text{RN}_\sigma \) and \( x \) occurs in the step term \( r_2 \) of type \( !(l \rightarrow \rho) \) for the safe type \( \rho := \sigma \rightarrow \sigma \). In either case, we conclude from the induction hypothesis and from the hypothesis on \( \tau \) being higher-type and non-safe that the second alternative of \( (\ast) \) is true of \( t \). This is obvious in all cases, except possibly where \( c = \text{RN}_\sigma \). Here observe \( \tau \prec !(l \rightarrow \rho) \iff \tau \prec !(l \rightarrow \rho) \iff \tau \preceq !l \) or \( \tau \preceq l \).

Case \( t = !r^\sigma \). So \( x \) occurs in \( r \), and the induction hypothesis on \( r \) yields either \( \tau \prec \sigma \), implying \( \tau \prec !\sigma \), or else \( \tau \preceq \rho \) for some \( y^\rho \in \text{FV}(r) = \text{FV}(t) \). In either case, \( (\ast) \) is true of \( t \).

Case \( t = \kappa\vec{s} \). We apply Kappa (5.4.1) to \( s_1 \) to obtain that \( t \) is of the form \( \kappa(\ldots \kappa(X\vec{r}_1)\vec{r}_2 \ldots)\vec{r}_n \) for some complete variable \( X \). In any case, the second alternative of \( (\ast) \) is true of \( t \). For either \( x = X \), or else \( x \) occurs in some \( r_{ij} \) of type \( \sigma_{ij} \), in which case the induction hypothesis gives either \( \tau \prec \sigma_{ij} \), implying \( \tau \prec \text{type}(X) \), or else \( \tau \preceq \rho \) for some \( y^\rho \in \text{FV}(r_{ij}) \subseteq \text{FV}(t) \).

\[ \square \]

Corollary 5.4.4 (Structure). Let \( t \) be a normal RA term of ground type or of type \( \sigma = \vec{\sigma} \rightarrow \tau \) where \( \vec{\sigma}, \tau \) all are safe or ground, and assume that \( \text{FV}(t) \) all are safe or ground.

(a) \( t \) contains safe or ground type variables only.

(b) Every subterm \( \kappa\tau \) of \( t \) is of the form \( \kappa\vec{s}y \) for some complete ground type variable \( y \).

(c) Every subterm \( rs \) of \( t \) has safe type.\(^2\)

\(^2\)Here we benefit from treating the recursors \( \text{RN}_\sigma \) as term former rather than as constants.
(d) If $\lambda x. r$ is a subterm of $t$, then $x$ is affinable in $r$, or else $x$ is ground.

Proof. Part (a) follows from Type Analysis (5.4.3), because the type of any higher-type non-safe variable would either occur strictly left in the type of $t$, or else it would occur strictly left in or be equal to the type of a free variable of $t$. In either case, this would contradict the hypothesis on $t$.

As for (b), given a subterm $\kappa r$ of $t$, then $\kappa r$ is normal (as $t$ is normal), and we may apply Kappa (5.4.1) to obtain that $r$ is of the form $\kappa(\ldots \kappa(X r_1) r_2 \ldots ) r_n$ for some complete variable $X$. Now part (a) implies that $X$ is a complete ground type variable $y$. Hence $\# r_1 = \ldots = \# r_n = 0$ and $\kappa r$ is $\kappa r y$.

As for (c), given a subterm $rs$ of $t$, then $rs$ is normal (as $t$ is normal), hence $rs$ is either of the form $U r$ with $U$ being a constant, or else it is of the form $\text{RN}_\sigma g h n s$. In either case, $rs$ is safe.

Part (d) follows simply from the hypothesis that $t$ is a normal RA term. Observe that this is the only place where we need that $t$ is in RA.

\[ \square \]

5.5. SRA-terms and the Bounding Theorem

In this section we will show that RA programs computing number-theoretical functions can be evaluated in polynomial time. In fact, Bounding Theorem 5.5.10 states that for every closed RA term $t$ of type $\bar{\sigma} \rightarrow \tau$ with $\bar{\sigma}$, $\tau$ all ground, one can find a polynomial $p_t$ such that for all numerals $\bar{n}$ of types $\bar{\sigma}$ one can compute the numeral $n f_t(\bar{n})$ in time $p_t(\sum | n_i |)$. Numerals are binary numerals preceded by any number of $!$'s.

To this end, we explore the effect of affinability for normal terms in order to implement within RA a kind of sharing concept. This gives rise to a subclass SRA providing a convenient control structure for evaluating RA programs $t$ at inputs $\bar{n}$ in polynomial time.

Recall that by Structure (5.4.4) normal RA programs computing a number-theoretical function enjoy a certain structure we call “simple”.

**Definition 5.5.1 (Simple terms).** A term is simple if it has safe or ground variables only, all subterms $\kappa s$ are ground, and all subterms $st$ are safe.

Suppose that $t$ is a simple RA term such that for every subterm $\lambda x. r$ with $x$ higher-type, the variable $x$ is affinable in $r$ — recall that by Structure (5.4.4) every normal RA program computing a number theoretical function is of that form. Then by repeated ground type $\beta$ expansions, viewed as a kind of sharing construct, we can obtain an equivalent simple term $\beta(t)$ in which each subterm $\lambda x. s$ with $x$ higher-type satisfies $\text{FO}(x, s) \leq 1$, that is, $x$ has at most one free occurrence in $s$. To obtain $\beta(t)$, we simply replace recursively every subterm $\lambda x. s$ with $s$ being of the form $\ldots u \ldots$ where $u$ is the minimal subterm of $s$ containing all occurrences of an affiliation of $x$ in $s$, $u = \ldots a \ldots a \ldots$ say, by $\lambda x. \ldots (\lambda y^1 \ldots y \ldots y \ldots )a \ldots$.

**Definition 5.5.2.** SRA-terms are simple RA terms $r$ such that $\text{(S)}$ every subterm $\lambda x. s$ with $x$ higher-type (safe) satisfies $\text{FO}(x, s) \leq 1$.

**Lemma 5.5.3 (Sharing).** If $t$ is a simple RA term such that for every subterm $\lambda x. s$ with $x$ higher-type, $x$ is affinable in $s$, then one can find an SRA-term $\beta(t)$ such that $\beta(t) \rightarrow^* t$.

Proof. By induction on the number of occurrences of bound higher-type variables. Consider an outermost subterm $\lambda x. r$ with $x$ higher-type and $\text{FO}(x, r) \geq 2$. By assumption $x$ has an affiliation of $a$ in $r$. Let $u$ be the minimal subterm of $r$ such that $u$ contains all occurrences of $a$ in $r$. Now let $t'$ result from $t$ by replacing $u$
with \((\lambda y.u[y/a])a\) for some new variable \(y\), where we write \(u[y/a]\) for the result of simultaneously replacing all occurrences of \(a\) in \(u\) with \(y\).

To apply the induction hypothesis to \(t'\), one must show that every affination in \(t\) inside \(\lambda x.r\) results in an affination in \(t'\). To see this, let \(\lambda z.s\) be a subterm of \(r\) such that \(z\) has an affination \(b\) in \(s\). If \(a\) has no occurrence in \(s\), then \(b\) is still an affination of \(z\) in \(t'\). Otherwise by scope equivalence, either all occurrences of \(a\) are in \(s\) and \(z \in \text{FV}(a)\), or else no occurrence of \(a\) in \(s\) has a free occurrence of \(z\). In the latter case, either \(a\) occurs in \(b\), in which case \(b[y/a]\) is an affination of \(z\) in \(t'\), or else \(a, b\) are separated, in which case \(b\) is still an affination of \(z\) in \(t'\). In the former case, the minimality of \(u\) implies that \(u\) is in \(s\). By construction \(t'\) results from \(t\) by replacing the subterm \(u\) of \(s\) with \((\lambda y.u[y/a])a\) for some new \(y\). Since \(z\) has a free occurrence in both \(a\) and \(b\), there are two cases. If \(a\) is a subterm of \(b\), then each occurrence of \(b\) contains exactly one occurrence of \(a\), for \(b\) is an affination of \(z\) in \(s\). By construction it follows that \(a\) is an affination of \(z\) in \(t'\). Otherwise if \(b\) is a subterm of \(a\), then by construction \(b\) is still an affination of \(z\) in \(t'\).

\[\square\]

**Note.** Every SRA-term \(t\) can be written uniquely in head form, being of the form \(U\bar{r}\) where \(U\) is a variable (safe or ground), a constant or \(!s, \kappa s\) (ground) or \(\text{RN}_s g h n\), or else \(U\) is of the form \(\lambda x.s\) with \(\text{FO}(x, s) \leq 1\), or else with \(\text{FO}(x, s) > 1\) and \(x\) ground. Call \(\bar{r}, s, x, U\) the components of \(t\).

One useful aspect of the subsystem SRA is that terms of certain higher types behave as if they were normal. This is entirely due to the built-in simplicity (5.5.1).

**Lemma 5.5.4 (Quasi-Normality).** Let \(t\) be an SRA-term.

(a) If \(t\) has higher type \(!\sigma\), then \(t\) is of the form \(!s\).

(b) If \(t\) has type \(!\lambda x . s\rightarrow\sigma\), then \(t\) is of the form \(\lambda x.^{!}\lambda x . s\rightarrow\sigma\).

(c) If \(t\) has type \(!(!\lambda x . s\rightarrow\sigma\)), then \(t\) is of the form \(\lambda x ^{!} . s\rightarrow\sigma\).

**Proof.** Obviously, (c) follows from (a), (b). As for (a), let \(U\bar{r}\) be the head form of \(t\). By simplicity (5.5.1) \(U\) cannot be a (higher-type) variable, for \(t\) has higher type \(!\sigma\). As well, \(U\) is neither a constant nor of the form \(\text{RN}_s g h n\), for all these have safe types. Furthermore, as \(t\) is higher-type, \(U\) cannot be \(\kappa s\), because by simplicity (5.5.1) \(\kappa s\) is ground, and so were \(t\). Finally, as \(t\) has type \(!\sigma\), \(U\) cannot be \(\lambda x . s\), because that would imply \#\(\bar{r}\) \geq 1, hence by simplicity (5.5.1) \(U\bar{r}_1\) were safe, and so were \(U\bar{r}\). Thus, the only possible form for \(U\) is \(!s\), implying \#\(\bar{r}\) = 0 and thus \(t = !s\) as required.

As for (b), let \(U\bar{r}\) be the head form of \(t\) with type \(!\lambda x . s\rightarrow\sigma\). By simplicity \(U\) is neither a variable nor of the form \(\kappa s\). Furthermore, for type reasons \(U\) is neither a constant nor of the form \(\text{RN}_s g h n\) or \(!r\). Thus, \(U\) must be \(\lambda x . s\), and in this case \#\(\bar{r}\) = 0 follows. For otherwise simplicity (5.5.1) would imply that \(U\bar{r}_1\) were safe, and so were \(U\bar{r}\), contradicting that \(t\) has type \(!\lambda x . s\rightarrow\sigma\).

\[\square\]

The system SRA is not closed under \(\text{RN}\) conversion and \(\beta\) conversions \((\lambda y.s)t\) with \(\text{FO}(y, s) \geq 2\) and \(t\) containing the unique free occurrence of a higher-type variable bound somewhere further up in the ambient term. But the structure of SRA and Quasi-normality (5.5.4) will enable us to “unfold” recursions and to perform within SRA all \(\beta\) conversions needed to compute normal forms.

Components of a head form are specified by numbering them in order from left to right. A general term formation is an operation on terms, resulting in the formation of a term \(tv, kt, !lt, (\lambda x.t)v, \) or \((t[s/x])v\), where \(t, x\) and \(s\) are any components of the given terms and \(v\) are optional trailing components of one of the given terms.
The algorithms $nf$ and $rf$ described below use a register machine model of computation, where each register may contain a term. One has an unlimited supply of registers $u, v, w$ etc. In particular, one associates a unique environment register $u_\times x$ with each variable $x$. A primitive computation step is any of the following operations: copying from one register to another; allocation of a new register and initialising it to a general term formation. In particular, each of the following takes one primitive step:

- test on the head form of $t$ and copy any component of $t$ into a register;
- test on $(\lambda x.s)r\vec{r}$ with $\text{FO}(x, s) > 1$ and formation of $r$ and $s\vec{r}$;
- test on $(\lambda x.s)r\vec{r}$ with $\text{FO}(x, s) \leq 1$ and form the term $(s[r/x])\vec{r}$;
- test on $c_r t_1 t_2 t_3 t_4\vec{r}$, and formation of $t_1$ and $t_j \vec{r}$ for some $j \in \{2, 3, 4\}$.

It can be easily seen that these operations can be simulated by a Turing machine in time polynomial in the lengths of the terms involved (cf. e.g. [19]).

**Definition 5.5.5.** An environment is a list $\vec{n}; \vec{x} := n_1, \ldots, n_k; x_1, \ldots, x_k$ where each $n_i$ is an SRA-term of the same type as $x_i$. A numeral environment is an environment $\vec{n}; \vec{x}$ such that each $n_i$ is a numeral.

**Theorem 5.5.6 (Base Evaluation).** For every RN-free SRA-term $t$ of ground type and numeral environment $\vec{n}; \vec{x}$ such that $\text{FV}(t) \subseteq \vec{x}$,

1. one can compute $nf(t[\vec{n}; \vec{x}])$ in at most $2|t|$ steps,
2. the number of used registers is $\leq |t| + \#\vec{n}$, and
3. every term $s$ occurring in the computation satisfies $|s| \leq |t| + \max |n_i|$.

**Proof.** We describe the algorithm $nf$, which at input $t, \vec{n}; \vec{x}$ outputs $nf(t, \vec{n}; \vec{x}) = nf(t[\vec{n}; \vec{x}])$ in the input register of $t$, by induction on $|t|$. For type reasons, $t$ is of the form $U\vec{r}$ where $U$ is either a variable $x_1$ or a constant, or $U$ is $!s$ or $\kappa s$, or else $U$ is $\lambda x.s$ with either $\text{FO}(x, s) \leq 1$, or else $\text{FO}(x, s) > 1$ and $x$ is ground.

If $t$ is $x_i$, then output $n_i$. We have performed two steps, and (2), (3) are obvious.

If $t$ is $0$, then output $0$. We have performed two steps, and (2), (3) are obvious.

If $t$ is $U\vec{r}$ where $U$ is one of the symbols $s_1, !$, first compute $n := nf(r, \vec{n}; \vec{x})$, then form $U n$. We have performed $\leq 2 + 2|r| \leq 2|t|$ steps, using $\leq 1 + |r| + \#\vec{n} \leq |t| + \#\vec{n}$ registers, and (3) follows.

If $t$ is $s_0\vec{r}$, first compute $n := nf(r, \vec{n}; \vec{x})$, then output $0$ if $n$ is $0$, otherwise form $s_0 n$. We have performed at most $3 + 2|r| \leq 2|t|$ steps, using at most $1 + |r| + \#\vec{n} \leq |t| + \#\vec{n}$ registers. Concerning (3), we obtain $|s_0 n| \leq 2 + |r| \leq \max |n_i| = |t| + \max |n_i|$.

If $t$ is $pr$, first compute $n := nf(r, \vec{n}; \vec{x})$, then form $m$ if $n$ is $s_i m$, else output $0$. Properties (1), (2), (3) follow as in the previous case.

If $t$ is $k\vec{r}$, first compute $\lambda n := nf(r, \vec{n}; \vec{x})$, then output $n$. We have performed $\leq 1 + 2|r| \leq 2|t|$ steps, using $\leq |r| + \#\vec{n}$ registers, and (3) follows directly from the induction hypothesis on $r$.

If $t$ is $c_\sigma s_0 t_1 t_2 \vec{r}$, first compute $n := nf(s, \vec{n}; \vec{x})$, then compute $nf(t_j \vec{r}, \vec{n}; \vec{x})$ where $j$ is $0$ if $n$ is $0$, and where $j$ is $1 + i$ if $n$ is of the form $s_i m$. We have performed $\leq 2 + 2|s| + 2|t_j \vec{r}| \leq 2|t|$ steps, using at most $1 + \max |s| + \#\vec{n}, |t_j \vec{r}| + \#\vec{n} \leq |t| + \#\vec{n}$ registers. (3) follows from the induction hypothesis on $t_j \vec{r}$.

If $t$ is $(\lambda x.s)r\vec{r}$ with $\text{FO}(x, s) > 1$, first compute $n := nf(r, \vec{n}; \vec{x})$, then copy $n$ to $ux$, and finally compute $nf(s\vec{r}, \vec{n}; \vec{x}, x)$. Observe that $\vec{n}; \vec{x}, x$ is a numeral environment such that $\text{FV}(s\vec{r}) \subseteq \vec{x}, x$. We have performed $\leq 2 + 2|r| + 2|s\vec{r}| \leq 2|t|$ steps, and (2), (3) follow as in the previous case.

If $t$ is $(\lambda x.s)r\vec{r}$ with $\text{FO}(x, s) \leq 1$, compute $nf((s[r/x])\vec{r}, \vec{n}; \vec{x}).$ As $|(s[r/x])\vec{r}| \leq |t|$, we have performed at most $1 + 2|(s[r/x])\vec{r}| \leq 2|t|$ steps, using $\leq |(s[r/x])\vec{r}| + \#\vec{n}$ registers, and (3) is obvious. \qed


Corollary 5.5.7 (Base Normalisation). For every closed RN-free SRA-term $t$ of ground type one can compute $nf(t)$ in at most $2|t|$ steps, using $\le|t|$ registers, and every term occurring in the computation has a length $\le|t|$.

In order to compute RN-free SRA-terms $rf(t, \vec{n}; \vec{x})$ in a given environment $\vec{n}; \vec{x}$, we slightly generalise the technique above. To “unfold” every recursion $R_{ghn}$ in $t$, we require that $FV(t[\vec{n}/\vec{x}])$ all are safe, because then by Completeness (5.3.1) $n[\vec{n}/\vec{x}]$ is closed, thus we may apply Base Normalisation (5.5.7) to compute the numeral $nf(rf(n[\vec{n}/\vec{x}]))$. Of course, this only works for RN-free $n$. To make the induction work, however, we require that $t$ is safe or ground. Finally, to ensure that $rf(t, \vec{n}; \vec{x})$ is in SRA, we show in addition that no safe variable $y \in FV(t) \setminus FV(\vec{n})$ is multiplied in $rf(t, \vec{n}; \vec{x})$. The overall guideline will be that we do as much as needed, but as little as possible.

Theorem 5.5.8 (RN-Elimination). Let $t$ be an SRA-term of safe or ground type. One can find a polynomial $q_t$ such that for all environments $\vec{n}; \vec{x}$ with $\vec{n}$ RN-free and $FV(t[\vec{n}/\vec{x}])$ all are safe, one can compute an RN-free SRA-term $rf(t, \vec{n}; \vec{x})$ satisfying $t[\vec{n}/\vec{x}] \rightarrow^* rf(t, \vec{n}; \vec{x})$ such that

(i) the number of steps, the number of used registers and the length of every term occurring in the computation all are $\le q_t(\sum |n_i|)$, and

(ii) for every safe variable $y \in FV(t) \setminus FV(\vec{n})$ one has $FO(y, rf(t, \vec{n}; \vec{x})) \le FO(y, t)$.

Proof. By induction of $|t|$. Let $\vec{n}; \vec{x}$ be a fixed but arbitrary environment with $\vec{n}$ all RN-free and $FV(t[\vec{n}/\vec{x}])$ all are safe. Let $m$ be $\sum |n_i|$. We write #steps, #registers and maxlen for the three quantities above, and call their maximum bound.

If $t$ is some $x_i$, then output $n_i$. We define $q_t := 2 + id$, and (i), (ii) are obvious.

If $t$ is $\lambda x.r$, first compute $R := rf(r, \vec{n}; \vec{x})$, then form $T := \lambda x.R$. Observe that both $x$ and $r$ are safe, for $t$ has safe type. Therefore, as $FV(t[\vec{n}/\vec{x}])$ all are safe, so are $FV(r[\vec{n}/\vec{x}])$, and we may apply the induction hypothesis. We define $q_t := |t| + q_r$, and (i) is obvious. As for (ii), by the induction hypothesis on $r$ no safe variable $y \in FV(r) \setminus FV(\vec{n})$ is multiplied in $R$, implying (ii) for $t$. It remains to show that $T$ is in SRA. As $\vec{n}$ is substitutable for $\vec{x}$ in $t$, we know $x \in FV(r) \setminus FV(\vec{n})$, hence $FO(x, rf(r, \vec{n}; \vec{x})) \le FO(x, r)$. Thus, as $t, R$ are in SRA, so is $T$.

If $t$ is $U r_1 \ldots r_t$ where $U$ is a variable distinct from all $\vec{x}$, or $U$ is one of the symbols $0, s, p, c, \kappa$, first compute each $R_i := rf(r_i, \vec{n}; \vec{x})$, then form $T := U R_1 \ldots R_t$. Observe that by Simplicity (5.5.1) each $r_i$ is safe or ground, and that $FV(r_i[\vec{n}/\vec{x}])$ all are safe. Therefore we can apply the induction hypothesis to define $q_t := |t| + \sum i q_{r_i}$. Property (i) is obvious, and (ii) for $T$ follows from (ii) for $R_1, \ldots, R_t$.

If $t$ is $(\lambda x.s) r r_1 \ldots r_t$, then we distinguish two subcases. But first observe that by simplicity (5.5.1) $(\lambda x.s)r$ is safe. Hence $s$ and each $r_i$ is safe, and $r$ is safe or complete ground. Now, as $FV(t[\vec{n}/\vec{x}])$ all are safe, so are $FV(r[\vec{n}/\vec{x}])$ and each $FV(r_i[\vec{n}/\vec{x}])$, and $FV(s[\vec{n}/\vec{x}])$ all are safe provided that $x$ is safe.

Subcase $x$ is safe. First compute $S := rf(s, \vec{n}; \vec{x})$, $R := rf(r, \vec{n}; \vec{x})$ and each $R_i := rf(r_i, \vec{n}; \vec{x})$, then form $T := (\lambda x.S) R R_1 \ldots R_t$. As above we see that $T$ is an SRA-term. Using the induction hypothesis, we define $q_t := |t| + q_s + \sum i q_{r_i}$. Then (i) is obvious, and (ii) for $T$ follows from (ii) for $S, R, R_1, \ldots, R_t$.

Subcase $x$ is complex. First compute $n := rf(r, \vec{n}; \vec{x})$, copy $n$ to $u_x$, then compute $rf(s r_1 \ldots r_t, n; \vec{x}, x)$, observe that $t' := s r_1 \ldots r_t$ is a safe SRA-term with $|t'| < |t|$ and $FV(t'[\vec{n}/\vec{x}])$ all safe. Hence we can apply the induction hypothesis to define $q_t := |t| + q_r + q_{t'} \circ (id + q_r)$. Property (i) is obvious. As for (ii), first observe that by Completeness (5.3.1) $r$ has no safe free variable, for $r$ is complete ground. Hence by (5.3.1)
Compute \(T \in \beta\) one obtains the numeral \(T(n, \vec{n}, \vec{x})\) as counter. If \(T\) holds \(n, \vec{n}, \vec{x}\), then (ii) for \(j\) rounds, compute \(T_j := (H, \vec{n}, \vec{x})\) with \(T_j\) obtained from \(n\) by deleting the first \(i\) leading constants \(s_0, s_1, \ldots, s_i\).

First we compute the numeral \(!n\). Since \(n\) has type \(\nu\), all free variables of \(n\) are complete. Hence \(n[\vec{n}/\vec{x}]\) is closed, since all free variables of \(t[\vec{n}/\vec{x}]\) are safe. Therefore, the induction hypothesis yields a closed RN-free SRA-term \(rf(n, \vec{n}; \vec{x})\) with bound \(\leq q_n(m)\), for \(n[\vec{n}/\vec{x}] \longrightarrow^* \) \(rf(n, \vec{n}; \vec{x})\). Then by Base Normalisation (5.5.7) one obtains the numeral \(!n := nf(rf(n, \vec{n}; \vec{x})) = nf(n[\vec{n}/\vec{x}])\) with

\[
\text{#steps} \leq 2|rf(n, \vec{n}; \vec{x})| \leq 2q_n(m), \quad \text{#registers} \leq q_n(m), \quad \text{maxlength} \leq q_n(m).
\]

We now compute the term \(T_0(T_1 \ldots (T_{k-1} G) \ldots)\) by an obvious loop with \(k \leq |n| \leq q_n(m)\) rounds. However, to obtain an estimate on our bound, we need to look into some details. Preparing the loop, copy \(H\) into a fixed register \(v\), compute \(G := rf(g, \vec{n}; \vec{x})\) with bound \(\leq q_g(m)\) in a result register \(u\), and consider the register \(w\) holding \(n = n_0\) as counter. If \(w\) holds \(n = 0\), output the content of register \(u\). Otherwise, \(w\) holds \(n \neq 0\) and \(u\) holds \((T_{k-1} \ldots (T_{k-1} G) \ldots)\). In that case compute \(!n_{k-1}\) from \(n\) and \(n_{k-1}\) in the environment register \(u_x\); this clearly is possible with some bound \(q_1(|n|) \leq q_1(q_n(m))\) for some polynomial \(q_1\). Compute \(T_{k-1-1} := rf(H, \vec{n}, \vec{n}_{k-1-1}; \vec{x}, x)\) in \(v\) with bound \(q_H(m + |n_{k-1-1}|) \leq q_H(m + q_n(m))\). Update \(u\) by applying the content of \(v\) onto \(u\)'s original content. This gives \(T_{k-1}(T_{k-1} \ldots (T_{k-1} G) \ldots)\) in one step, with no additional register and maxlength increased by \(|T_{k-1}| \leq q_H(m + q_n(m))\). Finally, update \(w\) to hold \(n_{k+1}\) and go to the initial test of the loop.

Let us now estimate our bound. We go \(k \leq |n| \leq q_n(m)\) times through the loop. From above we obtain:

\[
\text{#steps in each round} \leq 1 + q_1(|n|) + q_H(m + |n_{k-1-1}|) + 2 \leq 1 + q_1(q_n(m)) + q_H(m + q_n(m)) + 2
\]

The number of registers used is 3 (for \(v, u, u_x\) plus \(q_1(q_n(m))\) to compute \(!n_{k-1}\) plus \(q_H(m + q_n(m))\) to compute \(T_{k-1-1}\), and the maximum length of a term is

\[
q_n(m) + q_H(m + q_n(m)) \cdot q_n(m) + q_g(m)
\]

Hence the total bound for this part of the computation is

\[
(3 + q_1(q_n(m)) + q_H(m + q_n(m))) \cdot q_n(m) + q_g(m).
\]

Finally, in an obvious loop with \(l\) rounds, compute \(R_j := rf(r_j, \vec{n}; \vec{x})\) with bound \(q_{r_j}(m)\), assuming \(u\) holds the term \((T_0(T_1 \ldots (T_{k-1} G) \ldots))R_1 \ldots R_{j-1}\), and update \(u\) by applying this term to \(R_j\).
Concluding (i), the total number of steps, used registers and lengths of used terms can be bounded by

\[ q_t(m) := |t| + 2q_n(m) + (3 + q_l(q_n(m)) + q_H(m + q_n(m))) \cdot q_n(m) + q_g(m) + \sum_{i=1}^{l} q_{r_i}(m). \]

It remains to verify (ii) for \( \text{nf}(t, \vec{n}; \vec{x}) \). By Completeness (5.3.1) no \( T_i \) contains a safe free variable, because \( h \) has complete type and \( H[\vec{n}, \vec{m}/\vec{x}] \rightarrow^* T_i \) by the induction hypothesis on \( H \). Therefore, (ii) follows from the induction hypothesis (ii) on \( g, r_1, \ldots, r_l \).

Essentially by combining Closure under Reduction (5.3.4), Sharing (5.5.3), RN-Elimination (5.5.8) and Base Normalisation (5.5.7), for each closed RA term \( t \) of type \( h \rightarrow \tau \) with \( h, \tau \) all ground, we obtain a polynomial-time-evaluation strategy which at input \( \vec{n} \) outputs the numeral \( \text{nf}(t, \vec{n}) \).

First we clarify on the role of safe input positions \( \sigma_i \) for the case where \( \tau \) is complete. Essentially due to Completeness (5.3.1), such input positions are redundant. This will be used in the proof of the Bounding Theorem in order to reduce the case where \( \tau \) is complete to that where \( \tau \) is safe (\( t \)).

**Lemma 5.5.9 (Redundancy).** Let \( t \) be any normal RA term of type \( \sigma_1, \ldots, \sigma_k \rightarrow \! \! \tau \) such that \( \text{FV}(t) \) and \( \sigma_1, \ldots, \sigma_k, \tau \) all are ground. Let \( \varphi, \varphi' \) be any environments differing only on safe free variables of \( t \), and let \( \vec{n}, \vec{m} \) be any numerals of types \( \sigma_1, \ldots, \sigma_k \) which differ only on safe components \( \sigma_i \). Then \( \text{nf}(t, \vec{n}) = \text{nf}(t, \vec{m}) \).

**Proof.** We proceed by induction on \( t \). By Structure (5.4.4) all variables of \( t \) are safe or ground, and all subterms \( \kappa \sigma \) are ground. Therefore, as \( t \) has complete type, \( t \) cannot be of the form \( U \vec{F} \) with \( U \) a variable or a constant or a symbol \( \kappa \). If \( t = \kappa \sigma \) or \( \lambda \sigma \), or else if \( t \) is a variable, then \( t \) is ground, and by Completeness (5.3.1) \( t \) has complete free variables only. So in this case \( \text{nf}(t, \vec{n}) = \text{nf}(t, \vec{m}) \) follows from the hypothesis on \( \varphi, \varphi' \).

As \( t \) is normal, the only remaining case is where \( t \) is of the form \( \lambda x. r \). Then \( r \) has complete type, hence by Completeness (5.3.1) \( r \) has complete free variables only. Therefore, using the induction hypothesis on \( r \) and the hypothesis on \( \varphi, \varphi' \) and \( \vec{n}, \vec{m} \), straightforward semantical reasoning verifies \( \text{nf}(t, \vec{n}) = \text{nf}(t, \vec{m}) \).

**Theorem 5.5.10 (Bounding).** Let \( t \) be a closed RA term of type \( \sigma_1, \ldots, \sigma_k \rightarrow \tau \), where \( \sigma_1, \ldots, \sigma_k, \tau \) all are ground. Then one can find a polynomial \( p_t \) such that for all numerals \( \vec{n}_1, \ldots, \vec{n}_k \) with types \( \sigma_1, \ldots, \sigma_k \) respectively, one can compute the numeral \( \text{nf}(t, \vec{n}) \) in time \( p_t(\sum_i |n_i|) \).

**Proof.** One must find a polynomial \( p_t \) such that for all numerals \( \vec{n} \) of types \( \sigma \), one can compute \( \text{nf}(t, \vec{n}) \) in time \( p_t(m) \) where \( m := \sum_i |n_i| \). Let \( \vec{x} \) be new variables of types \( \sigma \). We consider two cases.

**Case \( \tau \) is safe.** Then, as \( t \) is an RA term, so is \( t \vec{x} \). The normal form of \( t \vec{x} \) is computed in a number of “steps” that might be large, but which is still only a constant with respect to \( \vec{n} \). By Closure under Reduction (5.3.4) and Structure (5.4.4) this is a simple RA term such that for every subterm \( \lambda x. r \) with \( x \) higher-type, the variable \( x \) is affinable in \( r \). Hence by Sharing (5.5.3) one obtains an SRA-term \( t' := \beta(\text{nf}(t, \vec{x})) \) satisfying \( t' \rightarrow^* \text{nf}(t, \vec{x}) \). Let \( c \) be the number of “steps” needed to compute \( t' \). By RN-Elimination (5.5.8) one obtains a closed RN-free SRA-term \( \text{nf}(t, \vec{m}; \vec{x}) \) such that \( t'[\vec{n}, \vec{m}/\vec{x}] \rightarrow^* \text{nf}(t, \vec{n}; \vec{x}) \). This requires at most \( q_{\text{nf}}(m) \) steps, and uses at most this many registers of this size. As the output is in a register, this also bounds the length \( |\text{nf}(t, \vec{m}; \vec{x})| \). Using Base Normalisation (5.5.7) one obtains the numeral \( \text{nf}(t, \vec{n}; \vec{x}) = \text{nf}(t, \vec{n}) \) in a total of \( c + 3q_{\text{nf}}(m) \) steps, using at most this many registers of this size. Since moving from our register machine computations to Turing machine computations requires only a \( p \)-time transformation, \( g_{\text{time}} \) say, we have thus computed the numeral \( \text{nf}(t, \vec{n}) \) in time \( p_t(m) := c' + 3q_{\text{nf}}(m) \cdot g_{\text{time}}(q_{\text{nf}}(m)) \) for some constant \( c' \).
Case $\tau$ is complete. By Redundancy (5.5.9) all safe input positions $\sigma_i$ of $t$ are redundant. This implies $[n_f(t\vec{n})] = [n_f(t\vec{n}')]$ where $\vec{n}'$ results from $\vec{n}$ by replacing each safe $n_i$ with 0. Thus, it suffices to compute $n_f(t\vec{n}')$. So let $t\vec{x}'$ be the RA term where $\vec{x}'$ results from $\vec{x}$ by replacing each safe $x_i$ with 0, and let $\vec{n}'', \vec{x}''$ result from $\vec{n}, \vec{x}$ respectively by cancelling all safe components. Then $[t\vec{n}''] = [t\vec{x}'[\vec{n}'']/\vec{x}''] = [t'\vec{n}''/\vec{x}'']$ where $t'$ is $\beta(n_f(t\vec{x}'))$. Therefore, the argument for the first case carries over to $t\vec{x}'$ with respect to $\vec{n}'', \vec{x}''$. In fact, one obtains the same bound $p_t$. \hfill $\Box$

5.6. Embedding the polynomial-time computable functions

In this section we complete the proof that RA programs compute exactly the polynomial-time computable functions. It remains to embed the $\text{FPTIME}$-functions into the system RA. One could use any of the resource-free function algebra characterizations [5, 6, 46] of $\text{FPTIME}$; we pick [5].

Theorem 5.6.1 (Embedding $\text{FPTIME}$). Every polynomial-time computable function is denoted by an RA term.

Proof. Recall (cf. Background 2.8) the characterisation of $\text{FPTIME}$ by a function algebra BC where functions come in the form $f(\vec{x}; \vec{y})$. We proceed by induction on the definition of $f(\vec{x}; \vec{y})$ in BC, associating to $f$ a closed RA term $t_f$ of type $\overrightarrow{t}; \overrightarrow{v} \rightarrow \iota$ such that $t_f$ is denoting $f$, that is, $[t_f] = f$.

Case $f$ is an initial function. If $f$ is 0, $s_i(; y)$ or $p(; y)$, then let $t_f$ be 0, $s_i$, $p$ respectively. If $f$ is the binary conditional $c(y_1, y_2, y_3)$, then let $t_f$ be $\lambda y_1 y_2 y_3.c(y_1 y_2 y_3)$. If $f$ is a projection $\pi_j^{m, n}$, then let $t_f$ be $\lambda x_1 \ldots x_{m+n}.u_j$ where $u_j$ is $\kappa x_j$ if $j \leq m$, otherwise $u_j$ is $x_j$.

Case $f(\vec{x}; \vec{y}) = g(\vec{g}(\vec{x}); h(\vec{x}; \vec{y}))$ with $\#\vec{g} = m$ and $\#\vec{h} = n$. Using the induction hypothesis, let $t_f$ be

$$\lambda \vec{x}\vec{y}.t_g[(t_{g_1}, \vec{x}) \ldots !(t_{g_m}, \vec{x})](t_{h_1}, \vec{x}\vec{y}) \ldots (t_{h_n}, \vec{x}\vec{y})$$

where $\vec{x}$ all are of type $!\iota$, and $\vec{y}$ all are of type $\iota$.

Case $f(0, \vec{x}; \vec{y}) = g(\vec{x}; \vec{y})$ and $f(s_i(x), \vec{x}; \vec{y}) = h_i(x, \vec{x}; \vec{y}, f(x, \vec{x}; \vec{y}))$ for $s_i(x) \neq 0$. So for $s_i(x) \neq 0$ we obtain $f(s_i(x), \vec{x}; \vec{y}) = h(s_i(x), \vec{x}; \vec{y}, f(x, \vec{x}; \vec{y}))$ where function $h$ satisfies

$$h(0, \vec{x}; \vec{y}, z) = 0$$

$$h(s_i(x), \vec{x}; \vec{y}, z) = h_i(x, \vec{x}; \vec{y}, z) \quad \text{for } s_i(x) \neq 0.$$ 

Using the induction hypothesis to obtain $t_{h_0}$ and $t_{h_1}$, we can define $h$ by cases:

$$t_h := \lambda x.\vec{y}.z. e \cdot c_i (\kappa x) 0 (t_{h_0}(\pi x)(\vec{x}\vec{y}z)) (t_{h_1}(\pi x)(\vec{x}\vec{y}z))$$

where $x, \vec{x}$ all are of type $!\iota$, and $\vec{y}, z$ all are of type $\iota$. Concluding the current case, we define $t_f$ by

$$\lambda \vec{x}.\vec{y}.\text{RN}_{\vec{x}\iota} (t_g[\vec{x}]) !(\lambda u^1 V^{\vec{x}\iota}.z. t_{h_0}(\vec{x}\vec{y}(V \vec{y}))) x.$$ 

In each case, one easily verifies $[t_f] = f$. \hfill $\Box$
6. Conclusion

In this thesis the impact of nesting control structures in programs where termination is guaranteed on the running time was investigated. We have given syntactical criteria that separate nesting of such control structures which do not cause a blow up in running time from those which might do. In case of imperative programming languages and ground type recursion, this gave rise to a uniform method, called measure $\mu$, which assigns in a purely syntactic fashion to each program $P$ in the language under consideration a natural number $\mu(P)$.

We proved that the measure $\mu$ on loop programs and on lambda terms over ground type variables enriched with primitive recursion characterises the Grzegorczyk hierarchy at and above the linear-space level. Furthermore, the measure $\mu$ on stack programs and on lambda terms over ground type variables enriched with recursion on notation characterises Turing machine computability with running time bounded by functions in the Grzegorczyk hierarchy at and above the linear-space level. In particular, we obtained that stack programs of $\mu$-measure 0 compute exactly the polynomial-time computable functions, and loop programs of $\mu$-measure 0 compute precisely the linear-space computable functions.

These results were put in perspective by showing that there are limitations intrinsic to any such method: No computable measure on stack programs can identify exactly those programs which run in polynomial time.

As for recursion, the method allows one to distinguish top recursions from side recursions (and flat recursions), and only the former are capable of blowing up the running time. In terms of imperative programs, it turned out that only loop statements with a top circle can cause a blow up in running time, while all other forms of loop statements do not.

Building on these insights, we designed a functional programming language $\text{RA}$ which allows one top recursion (on notation) in any type, and as many side recursions in any type as desired. In fact, we proved that $\text{RA}$ programs compute exactly the polynomial-time computable functions.

Thus, although one cannot hope to identify all programs of interest, the obtained results are promising in two directions: One hope is that the measure $\mu$ can be extended to actual programming languages. It already operates on programs written in a simple close-to-machine programming language which nevertheless has the flavour of realistic imperative languages. One can easily extend the control analysis to any non-size-increasing primitive operation. More costly is it to account for assignment statements $X := Y$ such that a great deal of programs which run in polynomial time can be identified, although assignment statements are freely used, essentially for copying tasks. Another extension concerns subprogram structure as common in many high level programming languages. The other direction concerns the hope that the characterisation of polynomial-time computability by recursion in all finite types is a first step towards making program extraction applicable in industry where safety has high priority. In fact, recent approaches (cf. Introduction) indicate that many other complexity classes inside polynomial time can be characterised by higher type recursion over appropriate ground data.

In any case, one should bear in mind that the method is constructive such that if it identifies a program with polynomial running time, then one can refine this information so as to derive automatically realistic bounds on the degree of the polynomial bounding the running time. Conversely, if the method yields super-polynomial running time, in contrast to a programmer’s expectation, the method always provides the information about the place in the program where this (possible) blow up in running time came about.

All in all, it might be a long way to go, but it is worthwhile doing it.
Bibliography


Index

$+^0$, 32
$+^q$, 32
$+^r$, 30
$A_n$, 13
$C$, 13
$E_n$, 14
$I$, 5, 10
$P$, 13
$S^n$, 30
$X−x$, 24
$H_σ$, 4, 9
$PR_1$, 24
$PR_2$, 24
$RA$, 6
$RN$, 24

smash, 17
$\text{VAR}(X)$, 24
$BC$, 21
$\text{bin}(x)$, 14
$A$, 5, 10
$C(\mathcal{P})$, 44, 53
$E^n$, 14
$E_n$, 12
$G^n$, 49
$I(\mathcal{P})$, 54
$L^n$, 49
$R_{i}^n$, 37
$R_{σ}(G, H)$, 9
$R_n$, 18
$U(\mathcal{P})$, 44, 53
$\mathcal{V}(\mathcal{P})$, 43
$E[n+1]$, 57
if $\text{top}(X) \equiv a \ [P]$, 43
$\llp$, 44
$\frac{p}{n}$, 44
deg, 18
$\text{FLINSPACE}$, 13

loop $X \ [P]$, 53
foreach $X \ [P]$, 43
FP, 17
FP-Bounding, 18
FP-Closure, 18
FP-TIME, 14
if $X \leq Y$ then $[P]$, 55
$X_{I1}$, 24
$Y_{I(x, X)}$, 27
$\dot{1}$, 15
$μ(\mathcal{P}) = n$, 45
$μ(t) = X; σ$, 24
$μ(t) = σ$, 25
$μ(t) = n$, 25
$nf(t; \vec{n}; \vec{x})$, 69
$nf(t)$, 12
$\text{nil}(X)$, 53
$ν(q)$, 32
$⊕$, 15
$⊗$, 15
$P \ll Q$, 47
$P_1; P_2$, 43, 53
$\text{pred}(X)$, 53
$R(τ)$, 24
$\text{RANK}(x, X)$, 24
$\rightarrow^{*}$, 12
$rf(t, \vec{n}; \vec{x})$, 70
$\text{rk}(\mathcal{P})$, 47
$σ \prec ρ$, 66
$\text{suc}(X)$, 53
$⊙$, 15
$\times^{r}$, 30
$[t]_{φ}$, 11
$\vec{n}; \vec{x}$, 69
$\overline{\text{nil}}; \vec{x}$, 69
$\vec{x}^0$, 32
$\{A\} P \{B\}$, 43
INDEX

loop, 53
nesting depth, 47
rank, 47
simple, 47
stack, 3, 43

Ramification, 5
Ranked types, 24
Ranked variable, 24

Recursion
allowable, 19
at rank \( l \), 23, 25
bounded on notation, 17
bounded primitive, 12
bounded simultaneous, 13
equations, 9
flat, 13, 23, 25
in all finite types, 4, 9–12
in type \( \sigma \), 4, 9
modified on notation, 17
nesting depth, 18
on notation, 15
safe, 21
side recursion, 2, 23, 25
simultaneous, 13
top recursion, 2, 23, 25

Redundant input position, 26

Statement
conditional, 43, 55
loop, 43, 53
sequence, 43, 53

System RA
\( \text{nf}(t, \vec{n}; \vec{x}) \), 69
RA terms, 62
\( \text{rf}(t, \vec{n}; \vec{x}) \), 70
SRA-terms, 67
components of head forms, 68
environment, 69
head form, 68
numeral, 67
numeral environment, 69

System \( T \), 9–12
\( \beta \)-conversion, 11
\( \text{FV}(t) \), 62
\( \text{nf}(t) \), 62

\( \mathbf{R}_\sigma \)-conversion, 12
\( r[s/\vec{x}^\sigma] \), 11
\( t \) reduces to \( t' \), 12
\( t[\vec{x}/\vec{x}] \), 12
\( t \longrightarrow^{*} t' \), 12
constants \( 0, S, \mathbf{R}_\sigma \), 10
constants \( 0, s_0, s_1, p, c_\sigma, \mathbf{RN}_\sigma \), 17, 61
conversion rules, 17, 24
denotational semantics, 11
normal form, 12
operational semantics, 11
strong normalisation, 12, 62
substitution, 11

Terms
binary numeral, 62
closed, 11
fair, 29
length, 62
marked, 61
normal form, 62
ramified affinable, 6
recursion, 61
redex, 62
reduct, 62
redundancy-free, 29
scope equivalent, 62
simple, 67
well-structured, 29

Types
, 4, 9
complete, 5, 61
ground, 61
higher, 61
incomplete, 5, 61
level, 10, 61
marked, 61
occurs strictly left, 66
safe, 5