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

26. November 2001

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 μ-measure $n+1$ compute exactly the functions in Grzegorczyk level $n+2$. In particular, programs of μ-measure 1 compute exactly the functions in 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 μ-measure 1 compute precisely the functions in FPTIME, the class of the functions computable on a Turing machine in time polynomial in the size of the input.

Another form of measure μ is obtained for the second kind of programs. It is shown that stack programs of μ-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 μ-measure 0 compute precisely the FPTIME functions. Furthermore, loop programs of μ-measure n compute exactly the functions in Grzegorczyk level $n+2$. In particular, loop programs of μ-measure 0 compute precisely the 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 RA supporting recursion on notation in all finite types. It is shown that RA programs compute exactly the FPTIME functions.

Contents

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 μ that assigns in a purely syntactic fashion to each program P in the language a natural number $\mu(\mathsf{P})$. It is shown that the measure μ 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 μ 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 μ-measure 0 compute exactly the polynomial-time computable functions, and loop programs of μ-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 μ, 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,

$$\begin{aligned} \mathrm{add}(0, y) &= y \\ \mathrm{add}(x+1, y) &= \mathrm{add}(x, y) + 1 \end{aligned}$$

consider the following definitions of multiplication and exponentiation:

$$\begin{aligned} \mathrm{mult}(0, y) &= 0 & \qquad \exp(0) &= 1 \\ \mathrm{mult}(x+1, y) &= \mathrm{add}(y, \mathrm{mult}(x, y)) & \qquad \exp(x+1) &= \mathrm{add}(\exp(x), \exp(x)) \end{aligned}$$

[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 mult runs in polynomial-time, while $\exp$ 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 mult passes its computed values to the "input position" y of add, which has no control over any other recursion, each computed value of $\exp$ controls the recursion 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

$$\text{input position } x_i \text{ controls } n_i \text{ "top recursions"}$$

"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 $f(\vec{x}, 0) = g(\vec{x})$ and $f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}))$. Assume inductively that

$$\begin{array}{ll} g(x_1, \ldots, x_l) & \text{has ranks } m_1, \ldots, m_l \\ h(x_1, \ldots, x_l, u, v) & \text{has ranks } n_1, \ldots, n_l, p, q \end{array}$$

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:

$$f \text{ is a } \begin{cases} \textit{top recursion} & \text{if } q \geq \max\{m_1, n_1, \ldots, m_l, n_l, p\} \\ \textit{side recursion} & \text{else} \end{cases}$$

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 add has ranks $1, 0$ and, therefore, the definition of mult has ranks $1, 1$, since it is a side recursion, while the definition of $\exp$ has rank 2, because it is a top recursion.

In fact, the primitive recursive programs of μ-measure 1, allowing *one* top recursion (like add) and *any number* of side recursions (like 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 μ 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 ι 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 $\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \ldots$ over a fixed but arbitrary alphabet. Such programs are built up from the usual instructions `push(a,X)`, `pop(X)`, `nil(X)` by sequencing, conditional statements `if top(X)`$\equiv$`a [Q]` and loop statements `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 `foreach X [P]`, we require that no instruction `push(a,X)`, `pop(X)` or `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 $\mathtt{P}'$ by simultaneously replacing each "free occurrence"of `X` in `P` (appearing as `if top(X)`$\equiv$`a [Q]`) with `U`. Then the sequence

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

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 `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 $\mathtt{P}_1$ which, without question, runs in polynomial time:

$$\mathtt{P}_1 :\equiv \mathtt{foreach}\ X_1\ \mathtt{[}\ldots\mathtt{foreach}\ X_l\ \mathtt{[push(a,Y)]]}$$

For if words $v_1, \ldots, v_l, w$ are stored in $\mathtt{X}_1, \ldots, \mathtt{X}_l, \mathtt{Y}$ respectively before $\mathtt{P}_1$ is executed, then `Y` holds the word $w\mathtt{a}^{|v_1|\cdot\ldots\cdot|v_l|}$ after the execution of $\mathtt{P}_1$. 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 $\mathtt{X}_i$ *controls* `Y` in the sense that `Y` is pushed inside the body of the loop governed by $\mathtt{X}_i$. But no "computed value" in the body of a loop of program $\mathtt{P}_1$ 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 `Y` controls some `Z`, and eventually `Z` controls `Y`. To see this, consider the following programs:

```
P2 :≡ nil(Y); push(a,Y); nil(Z); push(a,Z);
      foreach X [nil(Z); foreach Y [push(a,Z); push(a,Z)];
                 nil(Y); foreach Z [push(a,Y)]]
P3 :≡ nil(Y); push(a,Y); nil(Z);
      foreach X [foreach Y [push(a,Z); push(a,Z)]; push(a,Y)]
```

Both programs have "loop nesting depth" 2, and at first glance, they look quite similar. But on inspection we see that $\mathtt{P}_2$ runs in exponential time, whereas $\mathtt{P}_3$ has polynomial running time. For if w is initially stored in `X`, then `Z` holds the word $a^{2^{|w|}}$ after $\mathtt{P}_2$ is executed, while $a^{|w|\cdot(|w|+1)}$ is stored in `Z` after the execution of $\mathtt{P}_3$. The reason is simply that in the body of the outermost loop of $\mathtt{P}_2$, first `Y` controls `Z`, and then `Z` controls `Y`, in the sense above. In contrast, there is no such circle in $\mathtt{P}_3$.

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 μ-measure of a primitive instruction is 0, the μ-measure of a sequence is the maximal μ-measure of its components, and decisively,

$$\mu(\texttt{foreach X [Q]}) := \begin{cases} \mu(\texttt{Q}) + 1 & \text{if Q is a sequence with a top circle} \\ \mu(\texttt{Q}) & \text{else} \end{cases}$$

where a sequence $\texttt{Q}_1\texttt{;}\ \ldots\ \texttt{;}\ \texttt{Q}_l$ has a *top circle* if there exists a component $\texttt{Q}_i$ of maximal μ-measure such that some $\texttt{Y}$ controls some $\texttt{Z}$ in $\texttt{Q}_i$, and $\texttt{Z}$ controls $\texttt{Y}$ in $\texttt{Q}_1\texttt{;}\ \ldots\ \texttt{;}\ \texttt{Q}_{i-1}\texttt{;}\ \texttt{Q}_{i+1}\texttt{;}\ \ldots\ \texttt{;}\ \texttt{Q}_l$.

One easily verifies for the examples above that $\texttt{P}_1, \texttt{P}_3$ are of μ-measure 0, while $\texttt{P}_2$ is of μ-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 μ-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 μ-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 ϵ_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[2]. 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: ι is a type called "ground type", and if σ, ρ are types, then so is the "function type" $(\sigma \to \rho)$. Each type σ has associated with it a set H_σ of *functionals of type* σ inductively defined by:

$$\begin{aligned} \mathrm{H}_\iota &:= \mathbb{N} \\ \mathrm{H}_{\sigma\to\rho} &:= \text{the set of all mappings } F\colon \mathrm{H}_\sigma \to \mathrm{H}_\rho \end{aligned}$$

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* σ from functionals G, H of types $\sigma, (\iota \to \sigma \to \sigma)$ respectively, denoted by $\mathcal{R}_\sigma(G, H)$, if F satisfies the following "recursion equations":

$$\begin{aligned} F(0) &= G \\ F(x+1) &= H(x, F(x)) \end{aligned}$$

It is understood that G, H are defined beforehand where, apart from recursion in any type, one can use *explicit definitions* of the form $F'(X_1, \ldots, X_k) = \mathrm{RHS}$, 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 $\iota \to \iota$; that is, one obtains $e = \mathcal{R}_{\iota\to\iota}(S, H)$ 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 $\mathcal{A}$ of the well-known Ackermann function, defined by

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

where for unary f, $f^{(k)}$ denotes the *kth 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 $(\iota \to \iota) \to \iota \to \iota$, 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 $\mathcal{A} = \mathcal{R}_{\iota\to\iota}(S, H)$ 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,* $\mathcal{A}(x)$ above, *of an outer recursion to computed functionals,* i.e. $I(x, \mathcal{A}(x), y)$, *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,

$$\text{input positions of } \begin{cases} \text{complete higher type} & \text{can be used in any non-linear way} \\ \text{incomplete higher type} & \text{can be used in a certain linear way only} \\ \text{type } !\iota & \text{control at most one top recursion} \\ \text{type } \iota & \text{control no recursion.} \end{cases}$$

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

(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 ι 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 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, 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 $\mathrm{spec}(\vec{x}, y)$ holds", where $\mathrm{spec}(\vec{x}, y)$ is a quantifier-free formula. The method then allows one to automatically extract from the internally represented proof d a program $\mathrm{ep}(d)$ that is correct with respect to the formalised specification, that is, $\mathrm{spec}(\vec{x}, \mathrm{ep}(d)(\vec{x}))$ 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 μ-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 μ-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 μ, 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 μ is naturally closed under composition, unlike the function algebra BC. Thirdly, the measure μ 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 μ is similar in spirit to the ranking ρ in [6], which operates on the usual definition schemes of the primitive recursive functions. The ranking ρ gives elegant characterisations of the same complexity classes discussed here. Compared to the measure μ, 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, ρ does require further initial functions. Furthermore, while the proof that the polynomial-time computable functions can be simulated by programs of ρ-measure 1 relies on the, in parts hard going, proof given in [5], the measure μ 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 μ 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 μ-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 μ-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 μ-measure 0 compute precisely the polynomial-time computable functions. Secondly, the measure μ 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 μ 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 μ 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 ρ 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^ω 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^ω, 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 It satisfying $\mathrm{It}(f,x,y) = f^{(|x|)}(y)$, whereas It is not BFF.

Historically, Simmons [59] (cf. Background 2.7) was the first to give syntactic restrictions for the use of recursion in type $\iota \to \iota$ 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 μ operating on lambda terms enriched with two modes of recursion: primitive recursion and recursion on notation. The results for the measure μ 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" $\mathbf{S}\ldots\mathbf{S0}$ built from constant symbols $\mathbf{0}$ for zero and $\mathbf{S}$ 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: ι is a type called "ground type", and if σ, ρ are types, then so is the "function type" $(\sigma \to \rho)$. By repeatedly decomposing the right-hand side of $\to$, any type $\sigma \neq \iota$ can be written uniquely as

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

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

$$\begin{aligned} \mathrm{H}_\iota &:= \mathbb{N} \\ \mathrm{H}_{\sigma\to\rho} &:= \text{the set of all mappings } f\colon \mathrm{H}_\sigma \to \mathrm{H}_\rho \end{aligned}$$

Observe that functionals of type $\sigma \to \rho \to \tau$ can be identified with mappings $f\colon \mathrm{H}_\sigma \times \mathrm{H}_\rho \to \mathrm{H}_\tau$. Thus, the notation $(\sigma_0, \sigma_1, \ldots, \sigma_{k-1} \to \iota)$ is also used, and one may think of a functional $F \in \mathrm{H}_\sigma$ 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 $\iota \to \sigma$ is defined by *recursion in type* σ from functionals G, H of types $\sigma, (\iota \to \sigma \to \sigma)$ respectively, denoted by $\mathcal{R}_\sigma(G, H)$, if F satisfies the following "recursion equations":

$$\begin{aligned} F(0) &= G \\ F(x+1) &= H(x, F(x)) \end{aligned}$$

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

$$\begin{aligned} \mathrm{add}(0, y) &= y \\ \mathrm{add}(x+1, y) &= \mathrm{add}(x, y) + 1 \end{aligned}$$

one obtains $\mathrm{add} = \mathcal{R}_{\iota\to\iota}(G,H)$ 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,

$$\begin{aligned}\mathrm{mult}(0,y) &= 0\\ \mathrm{mult}(x+1,y) &= \mathrm{add}(y,\mathrm{mult}(x,y))\end{aligned}$$

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

Now, observe that by a single higher type recursion one can define functions of exponential growth. For example, writing the equations

$$\begin{aligned}e(0,y) &= 1+y\\ e(x+1,y) &= e(x,e(x,y))\end{aligned}$$

as recursion in type $\iota\to\iota$, one obtains $e = \mathcal{R}_{\iota\to\iota}(S,H)$ 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

$$\begin{aligned}\mathcal{A}(0)(y) &= 1+y\\ \mathcal{A}(x+1)(y) &= \mathcal{A}(x)^{(y+1)}(y)\end{aligned}$$

where for unary f, $f^{(k)}$ denotes the *kth 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 $I = \mathcal{R}_{(\iota\to\iota)\to\iota\to\iota}(G,H)$ 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 $\mathcal{A} = \mathcal{R}_{\iota\to\iota}(S,\tilde{H})$.

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 ε_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 ε_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 $\mathcal{R}_\sigma(G,H)$. 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 σ, denoted $s^\sigma, r^\sigma, t^\sigma$, are inductively built from *variables* x^σ and *constant symbols*

$\mathbf{0}$	of type ι	(for zero)
$\mathbf{S}$	of type $\iota\to\iota$	(for successor S)
$\mathbf{R}_\sigma$	of type $\sigma\to(\iota\to\sigma\to\sigma)\to\iota\to\sigma$	(for recursion in type σ)

by *lambda abstraction* $(\lambda x^\sigma.r^\rho)^{\sigma\to\rho}$ and *application* $(r^{\sigma\to\rho}s^\sigma)^\rho$.

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

An occurrence of a variable x^σ 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^σ in r are called *bound*. Terms without free occurrences of variables are called *closed*.

Intuitively, closed terms t^σ denote functionals of type σ. To make the construction work, however, one must define for arbitrary terms t *the functional* t *denotes in an environment* φ, 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 $\bigcup_\sigma \mathrm{H}_\sigma$. Given φ, $[\![t]\!]_\varphi$ is inductively defined by:

$$\begin{aligned}
[\![x^\sigma]\!]_\varphi &:= \varphi(x^\sigma)\\
[\![\mathbf{0}]\!]_\varphi &:= 0\\
[\![\mathbf{S}]\!]_\varphi &:= S\\
[\![\mathbf{R}_\sigma]\!]_\varphi(g,h) &:= \mathcal{R}_\sigma(g,h) \ \text{ for } g\in \mathrm{H}_\sigma, h\in \mathrm{H}_{\iota\to\sigma\to\sigma}\\
[\![r^{\sigma\to\rho}s^\sigma]\!]_\varphi &:= [\![r]\!]_\varphi([\![s]\!]_\varphi)\\
[\![\lambda x^\sigma.r]\!]_\varphi(f) &:= [\![r]\!]_{\varphi[x\leftarrow f]} \ \text{ for } f\in \mathrm{H}_\sigma
\end{aligned}$$

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

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

$$\begin{aligned}
t_{\mathrm{add}} &:= \lambda x^\iota y^\iota.\mathbf{R}_\iota\, y\,(\lambda u^\iota v^\iota.\mathbf{S}v)\,x\\
t_{\mathrm{mult}} &:= \lambda x^\iota y^\iota.\mathbf{R}_\iota\,\mathbf{0}\,(\lambda u^\iota v^\iota.t_{\mathrm{add}}\,y\,v)\,x\\
t_e &:= \mathbf{R}_{\iota\to\iota}\,\mathbf{S}\,(\lambda u^\iota V^{\iota\to\iota}y^\iota.V(Vy))\\
t_{\mathcal{A}} &:= \mathbf{R}_{\iota\to\iota}\,\mathbf{S}\,(\lambda u^\iota V^{\iota\to\iota}y^\iota.\,\mathbf{R}_\iota\,y\,(\lambda u^\iota v^\iota.Vv)\,(\mathbf{S}y))
\end{aligned}$$

Computation in system T is given by an *operational semantics*, roughly speaking a method of transforming a given term t into a representation $\mathsf{nf}(t)$ 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^σ may or may not have free occurrences in r^ρ, we define an input position, x^σ, of the functional that r represents. Given a "functional" s^σ, 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^σ 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 β*-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 σ as an operation on terms, one obtains the following $\mathbf{R}_\sigma$*-conversion* rules:

$$\mathbf{R}_\sigma\, g\, h\, \mathbf{0} \mapsto g$$
$$\mathbf{R}_\sigma\, g\, h\, (\mathbf{S}n) \mapsto h\, n\, (\mathbf{R}_\sigma g h n)$$

Observe that due to the binding mechanism (lambda abstraction), the type of a constant $\mathbf{R}_\sigma$ is minimal.

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

This follows from a general result usually referred to as "strong normalisation" stated below. We write $t \longrightarrow t'$ if t' results from t by converting one subterm of t according to $\mapsto$. We say t *reduces to* t' if $t \longrightarrow^* t'$ where $\longrightarrow^*$ denotes as usual the reflexive and transitive closure of $\longrightarrow$. Observe that $\longrightarrow$ is **correct**, that is, if $t \longrightarrow^* t'$ then t, t' denote the same functional in every environment. Furthermore, $\longrightarrow$ is **consistent with substitution**, that is, if $t \longrightarrow^* t'$ and $\vec{s} \longrightarrow^* \vec{s}'$ (component-wise), then $t[\vec{s}/\vec{x}] \longrightarrow^* t'[\vec{s}'/\vec{x}]$. 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 $t = t_1 \longrightarrow t_2 \longrightarrow \ldots$ 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 $\mathsf{nf}(t)$ for the normal form of t.

So we see that ι is a data type and normalisation allows one to produce data from programs on input data $\vec{\mathbf{n}}$, 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 $(\mathcal{PR})$ by a hierarchy of classes $\mathcal{E}_n$. Following a more modern approach from Ritchie [52], the functions at Grzegorczyk level n are built from the *initial functions* $0, S$, all projections $\Pi_i^m(\vec{x}) = x_i$ 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 $\vec{x}$,

$$f(\vec{x}) = g(h_1(\vec{x}), \ldots, h_l(\vec{x})).$$

- $f(x, \vec{y})$ 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.

Thus, the idea in the design of each class $\mathcal{E}_n$ 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:

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

Note that $A_1(x,y) = x+y$, $A_2(x,y) = x \cdot y$, $A_3(x,y) = x^y$, $A_4(x,y) = x^{x^{\cdot^{\cdot^{x}}}}$ 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 $f \in \mathcal{E}_n$, 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 $\mathcal{E}_0$ have "constant growth", functions in $\mathcal{E}_1$ have "linear growth", $\mathcal{E}_2$ 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, $\mathcal{E}_3$ 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:

$$\begin{aligned} P(0) &= 0 & \qquad C(0,y,z) &= y \\ P(x+1) &= x & C(x+1,y,z) &= z \end{aligned}$$

These are examples of *flat recursions*, that is, recursions which do not use any of their computed values. Observe that the predecessor is in $\mathcal{E}_0$, but the conditional is a proper element of $\mathcal{E}_1$. The latter implies that each class $\mathcal{E}_{n+1}$ 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 $\vec{x}, y$,

$$\begin{aligned} f_i(\vec{x},0) &= g_i(\vec{x}) \\ f_i(\vec{x},y+1) &= h_i(\vec{x},y,f_1(\vec{x},y),\ldots,f_k(\vec{x},y)). \end{aligned}$$

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 $\mathcal{E}_{n+2}$ is closed under bounded simultaneous recursion. But observe that one application of unbounded simultaneous recursion from functions in $\mathcal{E}_{n+2}$ yields functions in $\mathcal{E}_{n+3}$.

Let $\mathrm{Rel}(\mathcal{E}_n)$ denote the set of relations $R \subseteq \mathbb{N}^k$ with *characteristic function* $c_R \colon \mathbb{N}^k \to \mathbb{N}$ in $\mathcal{E}_n$, 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 $\mathrm{Rel}(\mathcal{E}_n)$ is closed under *propositional logic* and *bounded quantification*, that is,

if R, Q are in $\mathrm{Rel}(\mathcal{E}_n)$, then so are $\neg R,\ R \wedge Q,\ R \vee Q$,

if $R \subseteq \mathbb{N}^{k+1}$ is in $\mathrm{Rel}(\mathcal{E}_n)$, 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:

$$E_1(x) := x^2 + 2$$
$$E_{n+2}(x) := E_{n+1}^{(x)}(2)$$

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 *nth 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, $\mathrm{bin}(x)$ for $x \in \mathbb{N}$ (with $\mathrm{bin}(0) = \varepsilon$, 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 $\mathrm{bin}(x)$ by the initial functions 0 and the *binary successors* s_0, s_1 satisfying:

$$\begin{aligned} s_0(x) &= 2 \cdot x && (\text{operation } \mathrm{bin}(x) \mapsto \mathrm{bin}(x)0 \text{ for } x \neq 0) \\ s_1(x) &= 2 \cdot x + 1 && (\text{operation } \mathrm{bin}(x) \mapsto \mathrm{bin}(x)1) \end{aligned}$$

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 $x, \vec{y}$,

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f(s_i(x), \vec{y}) &= h_i(x, \vec{y}, f(x, \vec{y})) \quad \text{for } s_i(x) \neq 0. \end{aligned}$$

Observe that $\mathrm{bin}(y) = b_l \ldots b_1$ with $b_l = 1$ implies $y = s_{b_1}(s_{b_2}(\ldots s_{b_l}(0)\ldots))$. Thus, for recursion on notation, the recourse is from $b_l \ldots b_1$ to $b_l \ldots b_2$ to $b_l \ldots b_3$ 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, *string concatenation* $\oplus$ satisfying $x \oplus y = x \cdot 2^{|y|} + y$, or in other words, $\mathrm{bin}(x \oplus y) = \mathrm{bin}(x)\mathrm{bin}(y)$, can be defined by the obvious recursion on notation (in the second argument):

$$\begin{aligned} x \oplus 0 &:= x \\ x \oplus s_i(y) &:= s_i(x \oplus y) \end{aligned}$$

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

$$\begin{aligned} x \otimes 0 &:= 0 \\ x \otimes s_i(y) &:= (x \otimes y) \oplus x \end{aligned}$$

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:

$$\begin{aligned} p(0) &:= 0 & c(0, y_0, y_1) &:= y_0 \\ p(s_i(x)) &:= x & c(s_i(x), y_0, y_1) &:= y_i \quad (\text{ for } s_i(x) \neq 0) \end{aligned}$$

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

$$\begin{aligned} \odot(0) &:= 0 \\ \odot(s_0(x)) &:= \odot(x) \oplus \odot(x) \\ \odot(s_1(x)) &:= s_1(\odot(x) \oplus \odot(x)) \end{aligned}$$

Note that $|\odot(x)| = x$, and hence $p(\odot(x)) = \odot(x \dot{-} 1)$ (where $\dot{-}$ as usual denotes *modified subtraction* with $x \dot{-} y = 0$ for $x \leq y$). Thus $\odot$ can be used to synchronise the number of steps in a recursion on notation on $\odot(x)$ 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 $0, s_0, s_1, \Pi_i^m$ by composition and recursion on notation, such that

$$(^*) \qquad \odot(f(\vec{x})) = f^*(\odot(\vec{x})) \qquad \text{for all } \vec{x}.$$

Thus, once $x \mapsto |x|$ is defined by recursion on notation, we obtain $f(\vec{x}) = |f^*(\odot(\vec{x}))|$ as required. The proof of (*) is by induction on the structure of $f(\vec{x})$. If f is $0, S, \Pi_i^m$, then let f^* be $0, s_1, \Pi_i^m$ respectively, for $\odot(0) = 0$ and $\odot(x+1) = 2(2^x - 1) + 1 = \mathbf{s}_1(\odot(x))$. 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 (*) is obvious in the case of composition. In case of primitive recursion, one proceeds by side induction on x showing $f^*(\odot(x, \vec{y})) = \odot(f(x, \vec{y}))$. The base case follows from $\odot(0) = 0$ and the induction hypothesis on g, for $f^*(\odot(0, \vec{y})) = g^*(\odot(\vec{y})) = \odot(g(\vec{y})) = \odot(f(0, \vec{y}))$. The step case is concluded as follows:

$$\begin{array}{rll} \odot(f(x+1, \vec{y})) & = \odot(h(x, \vec{y}, f(x, \vec{y}))) & \text{by definition} \\ & = h^*(\odot(x, \vec{y}, f(x, \vec{y}))) & \text{by the I.H. on } h \\ & = h^*(\odot(x, \vec{y}), f^*(\odot(x, \vec{y}))) & \text{by the side I.H. on } x \\ & = f^*(\odot(x+1, \vec{y})) & \text{by } p(\odot(x+1)) = \odot(x) \end{array}$$

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

$$\begin{array}{rlrll} 0+1 & = s_1(0) & |0| & = 0 & \\ 2x+1 & = s_1(x) & |2x| & = |x| + 1 & (\text{ for } x \neq 0) \\ (2x+1)+1 & = s_0(x+1) & |2x+1| & = |x| + 1 & \end{array}$$

Conversely, recursion on notation can be reduced to primitive recursion. To see this, first observe that the initial functions $0, s_0, s_1 . \Pi_i^n$ are primitive recursive, and so are even and p, since

$$\begin{array}{rlrl} \mathrm{even}(0) & = 1 & p(0) & = 0 \\ \mathrm{even}(x+1) & = C(\mathrm{even}(x), 1, 0) & p(x+1) & = C(\mathrm{even}(x+1), p(x), p(x)+1). \end{array}$$

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 $x \dot{-} y = P^{(y)}(x)$ is primitive recursive, then so are the functions $c_{\leq}$, $\mathrm{odd}(x) = 1 \dot{-} \mathrm{even}(x)$, and $\mathsf{e}(x, y) := p^{(|x| \dot{-} y)}(x)$. 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 $\hat{f}(\vec{x}, y, z) = f(\vec{x}, \mathsf{e}(y, z))$ for $z \leq |y|$:

$$\hat{f}(\vec{x}, y, 0) = g(\vec{x})$$

$$\hat{f}(\vec{x}, y, z+1) = \begin{cases} h_0(\vec{x}, \mathsf{e}(y, z), \hat{f}(\vec{x}, y, z)) & \text{if } z+1 \leq |y| \wedge \mathrm{even}(\mathsf{e}(y, z+1)) \\ h_1(\vec{x}, \mathsf{e}(y, z), \hat{f}(\vec{x}, y, z)) & \text{if } z+1 \leq |y| \wedge \mathrm{odd}(\mathsf{e}(y, z+1)) \\ \hat{f}(\vec{x}, y, z) & \text{else} \end{cases}$$

By induction on z, one easily obtains $\hat{f}(\vec{x}, y, z) = f(\vec{x}, \mathsf{e}(y, z))$ for $z \leq |y|$, using $p(\mathsf{e}(y, z+1)) = \mathsf{e}(y, z)$ in the step case. This implies $\hat{f}(\vec{x}, y, z) = f(\vec{x}, y)$ for $z \geq |y|$, and hence $f(\vec{x}, y) = \hat{f}(\vec{x}, y, |y|)$, 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 *modified recursion on notation* from g, h, i.e.,

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f(s_i(x), \vec{y}) &= h(s_i(x), \vec{y}, f(p(x), \vec{y})) \quad \text{for } s_i(x) \neq 0. \end{aligned}$$

Based on modified recursion on notation, one obtains a "binary version" of system T by adding to the simply typed lambda calculus the following *constant symbols*:

$\mathbf{0}$	of type ι	(for zero)
$\mathbf{s}_0$	of type $\iota \to \iota$	(for binary successor s_0)
$\mathbf{s}_1$	of type $\iota \to \iota$	(for binary successor s_1)
$\mathbf{p}$	of type $\iota \to \iota$	(for binary predecessor p)
$\mathbf{c}_\sigma$	of type $\iota \to \sigma \to \sigma \to \sigma \to \sigma$	(for binary cases in type σ)
$\mathbf{RN}_\sigma$	of type $\sigma \to (\iota \to \sigma \to \sigma) \to \iota \to \sigma$	(for recursion on notation in type σ)

The denotational semantics for this version of system T is then as expected. In terms of the *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 *binary numeral* is either $\mathbf{0}$ or of the form $\mathbf{s}_{i_1} \ldots \mathbf{s}_{i_k} \mathbf{s}_1 \mathbf{0}$, where $i_j \in \{0, 1\}$ for $j = 1, \ldots, k$, $k \geq 0$. In the conversion rules below we assume that $\mathbf{s}_i \mathbf{n}$ is a binary numeral (distinct from $\mathbf{0}$). The *conversion rules* for the constant symbols above are then as expected:

$$\begin{aligned} \mathbf{s}_0 \mathbf{0} &\mapsto \mathbf{0} & \mathbf{c}_\sigma \mathbf{0}\, r\, t_0\, t_1 &\mapsto r \\ \mathbf{p} \mathbf{0} &\mapsto \mathbf{0} & \mathbf{c}_\sigma (\mathbf{s}_i \mathbf{n})\, r\, t_0\, t_1 &\mapsto t_i \\ \mathbf{p} (\mathbf{s}_i \mathbf{n}) &\mapsto \mathbf{n} & \mathbf{RN}_\sigma g\, h\, \mathbf{0} &\mapsto g \\ & & \mathbf{RN}_\sigma g\, h\, (\mathbf{s}_i \mathbf{n}) &\mapsto h\, (\mathbf{s}_i \mathbf{n})\, (\mathbf{RN}\, g\, h\, \mathbf{n}) \end{aligned}$$

2.5. Cobham's class FP

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

The first such restriction is from Cobham [16] who showed FPTIME = FP where functions in FP are inductively defined from the *initial functions* $0, s_0, s_1$, all projections π_i^n and $\mathrm{smash}(x, y) = 2^{|x| \cdot |y|}$ by composition and bounded recursion on notation, that is, $f(x, \vec{y})$ is defined by *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 $x, \vec{y}$:

$$\begin{aligned} f(0, \vec{y}) &= g(\vec{y}) \\ f(s_i(x), \vec{y}) &= h_i(x, \vec{y}, f(x, \vec{y})) \quad \text{for } s_i(x) \neq 0 \\ f(x, \vec{y}) &\leq b(x, \vec{y}) \end{aligned}$$

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" 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-**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 satisfying $f_q(\vec{x}) = 2^{q(|\vec{x}|)}$ is in FP, referred to as FP-**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 $q(\vec{x})$ is $q_1(\vec{x}) \circ q_2(\vec{x})$ with $\circ \in \{+,\cdot\}$. First recall that concatenation $\oplus$ satisfies $|x \oplus y| = |x| + |y|$. Hence $x \oplus y \leq 2^{|x|+|y|} \leq \mathrm{smash}(s_1(x), s_1(y))$, implying that $\oplus$ can be defined in FP by bounded recursion on notation. Thus, we obtain $f_q \in$ FP from the induction hypothesis on q_1, q_2, because we have $|p(2^x)| = x$, implying

$$2^{q_1(|\vec{x}|)+q_2(|\vec{x}|)} = \mathrm{smash}(1, p(2^{q_1(|\vec{x}|)}) \oplus p(2^{q_2(|\vec{x}|)}))$$
$$2^{q_1(|\vec{x}|)\cdot q_2(|\vec{x}|)} = \mathrm{smash}(p(2^{q_1(|\vec{x}|)}), p(2^{q_2(|\vec{x}|)})).$$

2.6. Heinermann's classes based on nesting depth of recursions

Heinermann [25] was the first to classify the $\mathcal{PR}$-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 *nth 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 ι, the *nesting depth* $\deg(t)$ of a term $t \in \mathsf{PR}_1$ is inductively defined by:

$$\begin{aligned} \deg(u) &:= 0 \ \text{ for } u \text{ among } x, \mathbf{0}, \mathbf{S} \\ \deg(\lambda x.r) &:= \deg(r) \\ \deg(rs) &:= \max\{\deg(r), \deg(s)\} \\ \deg(\mathbf{R}_\iota\, g\, h) &:= \max\{\deg(g), 1 + \deg(h)\} \end{aligned}$$

Under these definitions, we can restate the nth Heinermann class as

$$\mathcal{R}_n = \{f \in \mathcal{PR} \mid f = [\![t]\!] \text{ for some closed } t \in \mathsf{PR}_1 \text{ with } \deg(t) \leq n\}.$$

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

$$\begin{aligned} \mathrm{add}(0, y) &= y \\ \mathrm{add}(x+1, y) &= \mathrm{add}(x, y) + 1 \end{aligned}$$

Now consider the following definitions of multiplication and exponentiation:

$$\begin{array}{ll} \mathrm{mult}(0,y) = 0 & \exp(0) = 1 \\ \mathrm{mult}(x+1,y) = \mathrm{add}(y, \mathrm{mult}(x,y)) & \exp(x+1) = \mathrm{add}(\exp(x), \exp(x)) \end{array}$$

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 $\iota \to \iota$

Simmons [59] was the first to give syntactic restrictions to the use of recursion in type $\iota \to \iota$ 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 $\iota \to \iota$.

Simmons' key observation was that the use of recursion in type $\iota \to \iota$ 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 $\mathcal{A}$:

$$\begin{aligned} \mathcal{A}(0)(y) &= 1 + y \\ \mathcal{A}(x+1)(y) &= \mathcal{A}(x)^{(y+1)}(y) = I(y+1, \mathcal{A}(x), y) \end{aligned}$$

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 $\mathcal{A}(x)$, and it is a pure parameter in the definition of this iteration.

This gave rise to a type level two functional algebra $\mathcal{S}(\Theta)$, 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, $\iota \to \iota$ dormant input positions respectively, and where $\Theta(\vec{x};\vec{y};\vec{f})$ is an arbitrary but fixed functional[2]. Using this bookkeeping, functionals in $\mathcal{S}(\Theta)$ are defined inductively by the *initial functionals* Θ, 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 $\mathrm{App}_{j,k}(\vec{x};\vec{y};\vec{f}) = f_k(j)$ 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

$$F(\vec{x};\vec{y};\vec{f}) = G(\ldots, A_i(\vec{x};;), \ldots; \ldots, B_j(\vec{x};\vec{y};\vec{f}), \ldots; \ldots, \lambda y.C_k(\vec{x};y;\vec{f}), \ldots)$$

- $F(\vec{x};\vec{y};\vec{f})$ is defined by *allowable recursion* from $G(\vec{x};\vec{y};\vec{f}), H(u,\vec{x};\vec{y},v;\vec{f})$ if

$$\begin{aligned} F(0,\vec{x};\vec{y};\vec{f}) &= G(\vec{x};\vec{y};\vec{f}) \\ F(x+1,\vec{x};\vec{y};\vec{f}) &= H(x,\vec{x};\vec{y},F(x,\vec{x};\vec{y};\vec{f});\vec{f}). \end{aligned}$$

[2]The presence of "Θ" 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 $F(x,\vec{x};\vec{y};\vec{f})$ can only be passed to dormant input positions (v) of the "step functional" $H(u,\vec{x};\vec{y},v;\vec{f})$. 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 $\lambda y.C_k(\vec{x};y;\vec{f})$ 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 Θ using recursion in type $\iota\to\iota$. But if Θ is a function $\Theta(\vec{x};\vec{y};)$ with ground type inputs only, then every F in $\mathcal{S}(\Theta)$ is primitive recursive in Θ, that is, it can be defined from Θ 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 $\iota\to\iota$:

$$\begin{aligned} F(0,y) &= A(;y;)\\ F(x+1,y) &= B(x;y;\lambda y.F(x,y)) \end{aligned}$$

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

$$\begin{aligned} \Pi_0(0,\vec{x};\vec{y};\vec{f}) &= 0\\ \Pi_0(x+1,\vec{x};\vec{y};\vec{f}) &= S_{\#(\vec{y})+1}(x,\vec{x};\vec{y},\Pi_0(x,\vec{x};\vec{y};\vec{f});\vec{f}) \end{aligned}$$

Hence allowable composition can be used to turn every dormant input position into an active one, that is, if $A(\vec{x};y,\vec{y};\vec{f})$ is in $\mathcal{F}(\Theta)$, then so is A^* satisfying $A^*(\vec{x},y;\vec{y};\vec{f})=A(\vec{x};y,\vec{y};\vec{f})$. 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 $\mathcal{A}$ is ruled out.

So Simmons' approach gives insight as to why recursion in type $\iota\to\iota$ 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

$$t_{\mathcal{A}} := \mathbf{R}_{\iota\to\iota}\,\mathbf{S}\,(\lambda u^{\iota}V^{\iota\to\iota}y^{\iota}.\,\mathbf{R}_{\iota}\,y\,(\lambda u^{\iota}v^{\iota}.Vv)\,(\mathbf{S}y))$$

(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 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:

$$\begin{aligned} x\otimes 0 &= 0 & \odot(0) &= 0\\ x\otimes s_i(y) &= (x\otimes y)\oplus x & \odot(s_0(x)) &= \odot(x)\oplus\odot(x) \qquad (\text{ for } x\neq 0)\\ & & \odot(s_1(x)) &= s_1(\odot(x)\oplus\odot(x)) \end{aligned}$$

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 $x,\vec{x},\vec{y}$,
$$\begin{aligned} f(0,\vec{x};\vec{y}) &= g(\vec{x};\vec{y}) \\ f(s_i(x),\vec{x};\vec{y}) &= h(x,\vec{x};\vec{y},f(x,\vec{x};\vec{y})) \quad \text{for } s_i(x) \neq 0. \end{aligned}$$

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

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 μ on λ terms over ground type variables and recursion

In this chapter, two subsystems $\mathsf{PR}_1, \mathsf{PR}_2$ of Gödel's T with only ground type variables and ground type recursion are considered, where PR_1 is based on primitive recursion and 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 μ on PR_1 is closely related to the Grzegorczyk hierarchy: Programs of μ-measure $n+1$ exactly compute the functions in Grzegorczyk level $n+2$. In particular, programs of μ-measure 1 exactly compute the FLINSPACE functions. The measure μ on PR_2 characterises the same hierarchy of classes, except that programs of μ-measure 1 precisely compute the FPTIME functions.

3.1. Intuition

The measure μ 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 τ_i is either a natural number n_i or ι, the idea being that:

- if τ_i is n_i, then input position x_i *controls* n_i *top recursions* ,
- if τ_i is ι, 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 τ, we define

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

Furthermore, we extend $\max$ so as to operate on ranks (τ, τ'), and we define its "strict version" smax:

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

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	has no rank (it has no input position at all)
Successor $S(x)$	has rank 0
Projection $\pi^l_i(x_1, \ldots, x_l) = x_i$	has ranks $\iota, \ldots, \iota, 0, \iota, \ldots, \iota$ (with 0 for the only non-redundant input position x_i)

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

$g(y_1, y_2)$	has ranks τ_1, τ_2
$h_1(x_1, \ldots, x_l)$	has ranks $\rho_1, \ldots, \rho_l$
$h_2(x_1, \ldots, x_l)$	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 τ_j is not ι, that is:

$f(x_1,\ldots,x_l)$ has ranks $\max(\mathrm{smax}(\tau_1,\rho_1),\mathrm{smax}(\tau_2,\delta_1)),\ldots,\max(\mathrm{smax}(\tau_1,\rho_l),\mathrm{smax}(\tau_2,\delta_l))$

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

- f is defined by *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

$$\begin{array}{ll} g(x_1,\ldots,x_l) & \text{has ranks } \tau_1,\ldots,\tau_l \\ h(x_1,\ldots,x_l,u,v) & \text{has ranks } \rho_1,\ldots,\rho_l,\rho_u,\rho_v \end{array}$$

If the rank ρ_v of the *critical input position* v is q, then f is called a *recursion at rank* q, and

$f(x_1,\ldots,x_l,y)$ has ranks $\max(\tau_1,\rho_1,q),\ldots,\max(\tau_l,\rho_l,q),\max(\rho_u,q+1)$

and we say:

$$\begin{array}{ll} f \text{ is a } \textit{top recursion} & \text{if } q \geq \max\{\tau_1,\rho_1,\ldots,\tau_l,\rho_l,\rho_u\} \\ f \text{ is a } \textit{side recursion} & \text{otherwise} \end{array}$$

Otherwise if the *critical* input position v is redundant, then f is called *flat recursion*, and

$f(x_1,\ldots,x_l,y)$ has ranks $\max(\tau_1,\rho_1),\ldots,\max(\tau_l,\rho_l),\mathrm{R}(\rho_u)$.

In either case of primitive recursion, each position x_i of g,h may "contribute" to the rank of *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 $\mathrm{add}(x,y)$ is a top recursion and has ranks $1,0$.

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

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

Now consider functions g,h defined by $g(x):=x$ and $h(x,u,v):=\mathrm{mult}(v,v)=v^2$. 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,

$$\begin{aligned} f(x,0) &= x \\ f(x,y+1) &= h(x,y,f(x,y)) = f(x,y)^2 \end{aligned}$$

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 PR_1 and PR_2

Terms in PR_1 and 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 ι, and if τ is a type, then so is $\iota \to \tau$. In other words, the objects under consideration are just functions over the natural numbers. *Terms* in PR_i, with their types, are:

$$\mathsf{PR}_1 := x \mid (\lambda x.r^\tau)^{\iota\to\tau} \mid (r^{\iota\to\tau} s^\iota)^\tau \mid \mathbf{0}, \mathbf{S} \mid (\mathbf{R}\, g^\iota\, h^{\iota,\iota\to\iota})^{\iota\to\iota}$$
$$\mathsf{PR}_2 := x \mid (\lambda x.r^\tau)^{\iota\to\tau} \mid (r^{\iota\to\tau} s^\iota)^\tau \mid \mathbf{0}, \mathbf{s}_0, \mathbf{s}_1 \mid (\mathbf{RN}\, g^\iota\, h_0^{\iota,\iota\to\iota}\, h_1^{\iota,\iota\to\iota})^{\iota\to\iota}$$

Observe that the *recursor* $\mathbf{RN}$ 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 $\mathbf{0}$ or $\mathbf{s}_{i_1} \ldots \mathbf{s}_{i_k}\mathbf{s}_1\mathbf{0}$ where $i_j \in \{0, 1\}$, and that $\mathbf{s}_i\mathbf{n}$ denote binary numerals distinct from $\mathbf{0}$. The *conversion rules* for $\mathbf{RN}$ are then as expected:

$$\mathbf{RN}\, g\, h_0\, h_1\, \mathbf{0} \mapsto g$$
$$\mathbf{RN}\, g\, h_0\, h_1\, (\mathbf{s}_i\mathbf{n}) \mapsto h_i\, \mathbf{n}\, (\mathbf{RN}\, g\, h_0\, h_1\, \mathbf{n})$$

3.3. The μ-measure on PR_1 and PR_2

To determine the μ-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 λ-abstracting 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 ι, and if σ is a ranked type, then so are $(\iota \to \sigma)$ and $(n \to \sigma)$ where $n \in \mathbb{N}$. We use τ and ρ, possibly with subscripts, to denote ι or any $n \in \mathbb{N}$.

Furthermore, we use the following abbreviations:

$$\begin{aligned}
\textsc{var}(X) &:= \{x \mid \exists k\colon (x, k) \in X\} && \text{variables of } X\\
X{-}x &:= \{(y, j) \in X \mid y \neq x\} && X \text{ minus } x\text{-part}\\
\textsc{rank}(x, X) &:= \max\{k \mid (x, k) \in X\} && \text{rank of } x \text{ in } X\\
X_{\uparrow l} &:= \{(y, \max(j, l)) \mid (y, j) \in X\} && \text{lifting of } X \text{ to } l\\
\textsc{r}(\tau) &:= \begin{cases} l & \text{if } \tau = l\\ 0 & \text{else}\end{cases}
\end{aligned}$$

We define t *has ranked free variables* X *and ranked type* σ, denoted $\mu(t) = X; \sigma$, by induction on the structure of $t \in \mathsf{PR}_i$.

Definition 3.3.1 (The μ-measure). $\mu(t) = X;\sigma$ is inductively defined as follows:

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

(ii) $\mu(\mathbf{0}) := \emptyset;\iota$, and $\mu(\mathbf{c}) := \emptyset;(0 \to \iota)$ for $\mathbf{c}$ among $\mathbf{S}, \mathbf{s}_0, \mathbf{s}_1$.

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

$$\mu(\lambda x.r) := \begin{cases} X{-}x;(\textsc{rank}(x,X) \to \sigma) & \text{if } x \in \textsc{var}(X) \\ X;(\iota \to \sigma) & \text{else.} \end{cases}$$

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

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

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

$$\mu(\mathbf{R}\, r\, s) := X_{\restriction l} \cup Y_{\restriction l};(k \to \iota)$$

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

(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

$$\mu(\mathbf{RN}\, r\, t_0\, t_1) := X_{\restriction l} \cup Y_{\restriction l} \cup Z_{\restriction l};(k \to \iota)$$

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

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

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

One can read from the definition that $\mu(t) = X;\sigma$ implies $\textsc{var}(X) \subseteq \mathsf{FV}(t)$. 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 μ 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 $\mathbf{R}\, r\, s$ (and similar for $\mathbf{RN}\, r\, t_0\, t_1$).

- 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:

$$\begin{aligned} \mathsf{P} &:= \mathbf{R}\,\mathbf{0}\,(\lambda uv.u) & \mathsf{p} &:= \mathbf{RN}\,\mathbf{0}\,(\lambda uv.u)\,(\lambda uv.u) \\ \mathsf{C} &:= \lambda xyz.\mathbf{R}\,y\,(\lambda uv.z)\,x & \mathsf{c} &:= \lambda xyz.\mathbf{RN}\,y\,(\lambda uv.y)\,(\lambda uv.z)\,x \\ & & \mathsf{C} &:= \lambda xyz.\mathbf{RN}\,y\,(\lambda uv.z)\,(\lambda uv.z)\,x \end{aligned}$$

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 ι are *redundant* (cf. [44]).

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

Lemma 3.3.4 (Redundancy). *For all terms $t \in \mathsf{PR}_i$ and lists of variables $\vec{x} := x_1, \ldots, x_p \supseteq \mathsf{FV}(t)$ with $\mu(\lambda\vec{x}.t) = (\tau_1, \ldots, \tau_p, \tau_{p+1}, \ldots \tau_q \to \iota)$, 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 $1 \le j \le p$ is redundant. If t is $\lambda y.r$, then the claim for t follows from the induction hypothesis on r with respect to $\vec{x}, y \supseteq \mathsf{FV}(r)$. So consider the case where t is rs, assuming $\mu(r) = X; (\tau, \tau_{p+1}, \ldots \tau_q \to \iota)$ and $\mu(s) = Y; \iota$. First note that

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

Subcase $\tau = l$. Hence $\mu(t) = X \cup Y_{\restriction l}; (\tau_{p+1}, \ldots \tau_q \to \iota)$. For every input position $j > p$ of $[\![\lambda\vec{x}.t]\!]$, the claim follows from the induction hypothesis on r for $j+1$. For $1 \le j \le p$ with $\tau_j = \iota$ we conclude $x_j \notin \mathrm{VAR}(X \cup Y_{\restriction l})$ 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 $\mu(t) = X; (\tau_{p+1}, \ldots \tau_q \to \iota)$ 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 (*). Thus, the claim for t follows from the induction hypothesis on r.

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

Corollary 3.3.5 (μ-Constants). *Every term t with $\mu(t) = \emptyset; \iota$ denotes a constant c, that is, $[\![t]\!]_\varphi = c$ for every environment. In that case we have $[\![t[\vec{\mathbf{0}}/\mathsf{FV}(t)]]\!] = c$, and every reduction sequence for $t[\vec{\mathbf{0}}/\mathsf{FV}(t)]$ ends with the numeral $\mathbf{S}^c\mathbf{0}$ (or with a binary numeral denoting c in the case of $t \in \mathsf{PR}_2$).*

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

In contrast to the approach in [46], [45], the current definition of the measure μ admits that the μ-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 $t \in \mathsf{PR}_1$ with μ-measure n defines a function in Grzegorczyk class $\mathcal{E}_{n+1}$.

There are two forms of subterms whose μ-measure dominates that of the overall term. One is caused by *redundant applications* rs where $\mu(r) = X;(\iota \to \sigma)$ and $\mu(r) < \mu(s)$. By Redundancy (3.3.4) one can replace s with $\mathbf{0}$. The other form could be dubbed *constant-depth recursion*, that is, applications rs where $\mu(r) = X;(l \to \sigma)$, $\mu(rs) < l$ and $\mu(s) = \emptyset;\iota$. 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 $\mu(+(\mathbf{SS0})) = (0 \to \iota)$.

In order to eliminate constant-depth recursions without increasing the μ-measure, we first show that the μ-measure is stable under β, η reductions. Essentially we need to know how μ 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 $x \in \mathrm{VAR}(X)$, the central technical term is

$$Y_{\uparrow(x,X)} := \{(y, \max(j,k) \mid (y,j) \in Y, (x,k) \in X\}$$

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 $y \in \mathrm{VAR}(Y)$ 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 $x \in \mathrm{VAR}(X)$ then $\mu(t[s/x]) = X{-}x \cup Y_{\uparrow(x,X)};\sigma$.*

(b) *If $x \notin \mathrm{VAR}(X)$ 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) $Y_{\uparrow(x,U\cup V)} = Y_{\uparrow(x,U)} \cup Y_{\uparrow(x,V)}$

(2) If $x \notin \mathrm{VAR}(U)$ then $Y_{\uparrow(x,U)} = \emptyset$.

(3) $(Y_{\uparrow(x,U)})_{\uparrow l} = Y_{\uparrow(x,U_{\uparrow l})}$

(4) $(U{-}x)_{\uparrow l} = U_{\uparrow l}{-}x$

(5) $(U \cup V)_{\uparrow l} = U_{\uparrow l} \cup V_{\uparrow l}$

(6) $(U_{\uparrow l})_{\uparrow l} = U_{\uparrow l}$

(7) $U_{\uparrow 0} = U$

We proceed by induction on the structure of t showing part (a), where we assume $x \in \mathrm{VAR}(X)$ for each case $\mu(t) = X;\sigma$. If t is x, 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 $y \notin \mathsf{FV}(s) \supseteq \mathrm{VAR}(Y)$ and thus $y \notin \mathrm{VAR}(Y_{\uparrow(x,U)})$.

Subcase $y \in \mathrm{VAR}(U)$, hence $\mu(t) = U{-}y;(\mathrm{RANK}(y,U) \to \rho)$. As $x \in \mathrm{VAR}(U)$, the induction hypothesis on r implies $\mu(t[s/x]) = (U{-}x \cup Y_{\uparrow(x,U)}){-}y;(\mathrm{RANK}(y, U{-}x \cup Y_{\uparrow(x,U)}) \to \rho)$. Since $y \notin \mathrm{VAR}(Y_{\uparrow(x,U)})$ and $x \neq y$, we conclude $\mu(t[s/x]) = (U{-}y){-}x \cup Y_{\uparrow(x,U-y)};(\mathrm{RANK}(y,U) \to \rho)$.

Subcase $y \notin \mathrm{VAR}(U)$, thus $\mu(t) = U;(\iota \to \rho)$. As $x \in \mathrm{VAR}(U)$ and $y \notin \mathrm{VAR}(U \cup Y_{\uparrow(x,U)})$, the induction hypothesis on r implies $\mu(t[s/x]) = U{-}x \cup Y_{\uparrow(x,U)};(\iota \to \rho)$, concluding the current case.

Case t is $r_1 r_2$ where $\mu(r_1) = U;(\tau \to \sigma)$ and $\mu(r_2) = V;\iota$. If $\tau = \iota$ then $\mu(t) = U;\sigma$ and $x \in \mathrm{VAR}(U)$, so the induction hypothesis on r_1 implies $\mu(t[s/x]) = U{-}x \cup Y_{\uparrow(x,U)};\sigma$. So suppose that $\tau = l$. Hence $\mu(t) = U \cup V_{\uparrow l};\sigma$ and we must show $\mu(t[s/x]) = (U \cup V_{\uparrow l}){-}x \cup Y_{\uparrow(x,U\cup V_{\uparrow l})};\sigma$.

Subcase $x \in \mathrm{VAR}(U) \setminus \mathrm{VAR}(V)$. By (1), (2) we must show (*) $\mu(t[s/x]) = (U{-}x) \cup V_{\uparrow l} \cup Y_{\uparrow(x,U)};\sigma$. The induction hypothesis on r_1 yields $\mu(r_1[s/x]) = (U{-}x) \cup Y_{\uparrow(x,U)};(l \to \sigma)$, and part (b) of the lemma gives $\mu(r_2[s/x]) = V;\iota$. Now (*) follows by definition.

Subcase $x \in \mathrm{VAR}(V) \setminus \mathrm{VAR}(U)$. By (1), (2) we must show (*) $\mu(t[s/x]) = U \cup (V_{\uparrow l} - x) \cup Y_{\uparrow(x,V_{\uparrow l})};\sigma$. We conclude $\mu(r_1[s/x]) = U;(l \to \sigma)$ from part (b), and the induction hypothesis on r_2 yields $\mu(r_2[s/x]) = (V{-}x) \cup Y_{\uparrow(x,V)};\iota$. Hence $\mu(t[s/x]) = U \cup (V{-}x)_{\uparrow l} \cup (Y_{\uparrow(x,V_{\uparrow l})})_{\uparrow l};\sigma$ by definition and (5). Now (*) follows from (4) and (3), (6).

Subcase $x \in \mathrm{VAR}(V) \cap \mathrm{VAR}(U)$. We have to show (*) $\mu(t[s/x]) = (U{-}x) \cup (V_{\uparrow l}{-}x) \cup Y_{\uparrow(x,U)} \cup Y_{\uparrow(x,V_{\uparrow l})};\sigma$ according to (1). The induction hypothesis yields $\mu(r_1[s/x]) = (U{-}x) \cup Y_{\uparrow(x,U)};(l \to \sigma)$ and $\mu(r_2[s/x]) = (V{-}x) \cup Y_{\uparrow(x,V)};\iota$. Hence $\mu(t[s/x]) = (U{-}x) \cup Y_{\uparrow(x,U)} \cup (V{-}x)_{\uparrow l} \cup (Y_{\uparrow(x,V)})_{\uparrow l};\sigma$ by definition and (5). Now (*) follows from (4) and (3), concluding the current case.

Finally, if t is a recursion $\mathbf{R}\, r_1\, r_2$ (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 μ-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 $z \notin \mathsf{FV}(r)$ and $z \neq y$. Suppose that $\mu(r) = U;\sigma$.

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

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

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

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

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

Proof. We proceed by induction on $t \longrightarrow_\beta t'$. The statement is obvious if t is $\mathbf{s_0 0}$ and $t' = \mathbf{0}$.

If t is $\lambda x.r$, then t' is $\lambda x.r'$ and $r \longrightarrow_\beta r'$. Hence we are done by the induction hypothesis on r.

If t is rs and t' is $r's$ with $r \longrightarrow_\beta r'$, or t' is rs' with $s \longrightarrow_\beta s'$, 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 $\mu(t) = X{-}x \cup Y_{\uparrow \text{RANK}(x,X)};\sigma$ by definition, and $\mu(t') = X{-}x \cup Y_{\uparrow(x,X)};\sigma$ by Substitution (3.3.6) (a). Now observe that $x \in \text{VAR}(X)$ implies $\text{VAR}(Y_{\uparrow(x,X)}) = \text{VAR}(Y_{\uparrow \text{RANK}(x,X)})$ and $\text{RANK}(y, Y_{\uparrow(x,X)}) = \max\{\text{RANK}(y,Y), \text{RANK}(x,X)\} = \text{RANK}(y, Y_{\uparrow \text{RANK}(x,X)})$ for all $y \in \text{VAR}(Y_{\uparrow(x,X)})$.

Finally, if t is $\mathbf{R}\, r\, s$ (similar for $\mathbf{RN}\, r\, h_0\, h_1$), 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 μ-measure is preserved when performing β reductions, unfolding constant-depth recursions $t := \mathbf{R}\, g\, h\, \mathbf{n}$ can decrease the μ-measure for various reasons. To see this, suppose that $\mu(g) = X;\iota$ and $\mu(h) = Y;(\tau_0, \tau_1 \to \iota)$. Then one obtains

$$\mu(t) = X_{\uparrow l} \cup Y_{\uparrow l};\iota \text{ where } l = \text{R}(\tau_1).$$

Obviously, if $\mathbf{n}$ is $\mathbf{0}$ then $t \longrightarrow^* g$ and $\mu(g) < \mu(t)$ might happen. So assume that $\mathbf{n}$ denotes a non-zero number, hence $t \longrightarrow^* t' := h\, \mathbf{n}{-}\mathbf{1}\, (\ldots (h\, \mathbf{0}\, g) \ldots)$. In that case there are two possibilities for $\mu(t') < \mu(t)$. One is when $\mathbf{R}\, g\, h$ is a flat recursion ($\tau_1 = \iota$), because then $\mu(t') = Y;\iota$. The other one is when $\mathbf{R}\, g\, h$ is a recursion at rank l ($\tau_1 = l$) and $\mathbf{n}$ is $\mathbf{S0}$; in this case t' is $h\, \mathbf{0}\, g$ with $\mu(t') = Y \cup X_{\uparrow l};\iota$, and so $\mu(t') < \mu(t)$ might happen for $X = \emptyset$. Observe that if $\mathbf{n}$ denotes a number ≥ 2, then $\mu(t') = X_{\uparrow l} \cup Y_{\uparrow l};\iota$.

It follows that if we want to eliminate constant-depth recursions, then we will have to unfold recursions and that will decrease the overall μ-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 $\mu(r) = X;(\iota \to \sigma)$.
- 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 $\mathsf{nf}(t)$ of t. □

The cost of computing the normal form $\mathsf{nf}(t)$ might be extremely high, because one already might need super-exponential time (in the size of t) to compute the β-normal form. But to obtain fair terms, one need not to fully normalise t but convert only some β-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 μ-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 μ-measure n can be

turned into a bounded recursion in $\mathcal{E}_{n+1}$. In fact, as side recursions do not increase the μ-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:

$$\begin{array}{rcll} S^1(x) & := & x+1 & \qquad S^1 := \mathbf{S} \\ 0 +^r y & := & y & \\ (x+1) +^r y & := & S^r(x +^r y) & \qquad +^r := \lambda xy.\mathbf{R}\, y\, (\lambda uv.S^r v)\, x \\ 0 \times^r y & := & 0 & \\ (x+1) \times^r y & := & y +^r (x \times^r y) & \qquad \times^r := \lambda xy.\mathbf{R}\, \mathbf{0}\, (\lambda uv.\, +^r yv)\, x \\ S^{r+1}(y) & := & S^r((y +^r y) \times^r (y +^r y)) & \qquad S^{r+1} := \lambda y.S^r(\times^r(+^r yy)\,(+^r yy)) \end{array}$$

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:

$$\begin{array}{rcl} \mu(S^r) & = & (r\text{–}1 \to \iota) \\ \mu(+^r) & = & (r, r\text{–}1 \to \iota) \\ \mu(\times^r) & = & (r, r \to \iota) \end{array}$$

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 $\circ \in \{+^s, \times^s, \max\}$ 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* $x' \to x' + 1$, 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:

$$\begin{array}{lll} (x' + 1) +^{s+1} y' & = & S^{s+1}(x' +^{s+1} y') \qquad \text{by definition} \\ & \geq & S^r(x'' +^r y) \qquad \text{by (*) and the side I.H.} \\ & = & x +^r y \qquad \text{by definition} \end{array}$$

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:

$$\begin{array}{lll} x +^r (y +^s z) & \leq & x +^s (y +^s z) \qquad \text{by (d)} \\ & = & S^{s(x+y)}(z) \qquad \text{by (a), applied two times} \\ & \leq & S^{s(x+^r y)}(z) \qquad \text{by (d)} \\ & = & (x +^r y) +^s z \qquad \text{by (a)} \end{array}$$

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, $y +^s y \leq S^r((y +^r y)^2) \leq S^r((y +^r y) \times^r (y +^r y)) \leq S^{r+1}(y)$. □

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 $q(\vec{x})$ to mean $\mathsf{Var}(q) \subseteq \{\vec{x}\}$. For arguments $\vec{a}$, the *value* $q(\vec{a})$ *of applying* q *to* $\vec{a}$ is defined as usual, that is, $c(\vec{a}) := c$, $x_i(\vec{a}) := a_i$ and $(q_1 \circ q_2)(\vec{a}) := q_1(\vec{a}) \circ q_2(\vec{a})$.

We want to prove that every function defined by a term t with μ-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 μ-measure 1.

To overcome this obstacle, we define a *modified measure* ν for Grzegorczyk polynomials only. Intuitively, ν is very much like μ, except considering $\max$ to be an initial function. The Modified Bounding lemma then states that every function represented by a term t with μ-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 μ coincide with the "input ranks" of q_t assigned by ν. 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 μ-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 μ, 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.

- $\nu(c,x) := \nu(y,x) := 0$
- $\nu(\max(q_1,q_2),x) := \max(\nu(q_1,x),\nu(q_2,x))$
- $\nu(q_1 \times^r q_2,x) := \begin{cases} \max(\nu(q_1,x),\nu(q_2,x),r) & x \in \mathsf{Var}(q_1) \cup \mathsf{Var}(q_2) \\ 0 & \text{else} \end{cases}$
- $\nu(q_1 +^r q_2,x) := \begin{cases} \max(\nu(q_2,x),r-1) & x \in \mathsf{Var}(q_2) \setminus \mathsf{Var}(q_1) \\ \max(\nu(q_1,x),\nu(q_2,x),r) & x \in \mathsf{Var}(q_1) \\ 0 & \text{else.} \end{cases}$

The *modified measure* of q, denoted $\nu(q)$, is defined by $\nu(q) := \max\{\nu(q,x) \mid x \in \mathsf{Var}(q)\}$.

Note that $x \notin \mathsf{Var}(q)$ 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. ν.

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 $q(\vec{x}) \leq c_q +^{\nu(q)+1} \max \mathsf{Var}(q)$.*

Proof. Induction on the structure of q, using $\vec{x} := \mathsf{Var}(q)$. 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:

$$\begin{aligned} q(\vec{x}) &\leq (c +^{r+1} \max \vec{x}) \circ (c +^{r+1} \max \vec{x}) && \text{by the I.H. and (3.4.2) (d)} \\ &\leq S^{r+1}(S^{r+1^{(c)}}(\max \vec{x})) && \text{by (3.4.2) (f), (a)} \\ &= (c+1) +^{r+1} \max \vec{x} && \text{by (3.4.2) (a)} \end{aligned}$$

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 $+ \equiv +^1$.

Lemma 3.4.8 (Top Level Separation). *For every Grzegorczyk polynomial q one can find a Grzegorczyk polynomial q' over $\vec{x}^q$ such that*

- $q(\vec{x}) \leq q'(\vec{x}^q) +^q \max \vec{x}^*$ *for some* $\vec{x}^* \subseteq \mathsf{Var}(q) \backslash \vec{x}^q$,
- $\nu(q', x) = \nu(q)$ *for every* $x \in \vec{x}^q$.

Proof. Induction on the structure of q. The cases where q is a constant or a variable are obvious, as $\vec{x}^q = \vec{x}$ and $q(\vec{x}) = q +^0 0$. So assume that q is $q_1(\vec{x}_1) \circ q_2(\vec{x}_2)$ with $\vec{x}_i := \mathsf{Var}(q_i)$ 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 $\vec{x}^q = \vec{x}_2^q$. Using the induction hypothesis on q_2, we define $q'(\vec{x}^q) := \max(c, q_2'(\vec{x}_2^q))$ to obtain $q(\vec{x}) \leq q'(\vec{x}^q) +^q \max(\vec{x}_1 \cup \vec{x}_2^*)$ 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 $q(\vec{x}) \leq q'(\vec{x}^q) +^q \max(\vec{x}_1^* \cup \vec{x}_2^*)$ for $q'(\vec{x}^q) := \max(q_1'(\vec{x}_1^q), q_2'(\vec{x}_2^q))$, using the induction hypothesis on q_1, q_2 and Lemma 3.4.2 (d).

Case $\circ$ is $+^q$. First observe that in this case $\nu(q, x) = \nu(q)$ for every $x \in \mathsf{Var}(q_1)$. If $\nu(q_2) < \nu(q)$ then we conclude $q_2(\vec{x}_2) \leq c +^q \max \vec{x}_2$ from Domination (3.4.6), and $\vec{x}^q = \vec{x}_1$. We define $q'(\vec{x}^q) := q_1(\vec{x}_1) +^q c$ to obtain $q(\vec{x}) \leq q'(\vec{x}^q) +^q \max \vec{x}_2$ by Lemma 3.4.2 (d), (e). Otherwise if $\nu(q_2) = \nu(q)$ then we obtain $q(\vec{x}) \leq q'(\vec{x}^q) +^q \max \vec{x}_2^*$ for $q'(\vec{x}^q) := q_1(\vec{x}_1) +^q q_2'(\vec{x}_2^q)$, by the I.H. on q_2 and Lemma 3.4.2 (d), (e).

Case $\circ$ is $\times^q$. Hence $\vec{x}^q = \vec{x}_1 \cup \vec{x}_2$, and $q' := q$ will do, since $q \leq q +^q 0$.

Case $\circ$ is $+^r$ or $\times^r$ 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 $q' := c +^q q''$ will do, where q'' is the Grzegorczyk polynomial obtained as above for $\max(q_1, q_2)$.

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

$$\nu(q(\vec{q}), y) = \max\{\nu(q, x_i), \nu(q_i, y) \mid x_i \in \mathsf{Var}(q), y \in \mathsf{Var}(q_i)\}.$$

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-$\mathsf{dcp}(q) = q_0(\vec{x}, w_1, \ldots, w_k), q_1(\vec{x}), \ldots, q_k(\vec{x})$ of Grzegorczyk polynomials, called the* s-decomposition of q, *such that $\vec{w} \in \mathsf{Var}(q_0)$ whenever $k > 0$, and*

(i) $q(\vec{x}) = q_0(\vec{x}, q_1(\vec{x}), \ldots, q_k(\vec{x}))$,

(ii) *if* $\nu(q, z) \leq s$ *then* $z \notin \bigcup \mathsf{Var}(q_{i+1})$,

(iii) *if* $z \in \bigcup \mathsf{Var}(q_{i+1})$ *then* $\nu(q, z) = \max\{\nu(q_1, z), \ldots, \nu(q_k, z)\}$,

(iv) $\nu(q_{i+1}, z) \geq s + 1$ *for* $z \in \mathsf{Var}(q_{i+1})$,

(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 $s\text{-}\mathsf{dcp}(q) := q, \emptyset$ will do. So consider the case where q is $q' \circ q''$. The induction hypothesis provides suitable s-decompositions $q'_0(\vec{x}, \vec{u}), q'_1(\vec{x}), \ldots, q'_k(\vec{x})$ and $q''_0(\vec{x}, \vec{v}), q''_1(\vec{x}), \ldots, q''_l(\vec{x})$, where $\vec{u} \cap \vec{v} = \emptyset$. If $\circ$ is $\max$, or if $\circ$ is $\times^r$ with $r \leq s$, or if $\circ$ is $+^r$ with $r \leq s+1$, then we define

$$s\text{-}\mathsf{dcp}(q) := q_0(\vec{x}, \vec{u}, \vec{v}), q'_1(\vec{x}), \ldots, q'_k(\vec{x}), q''_1(\vec{x}), \ldots, q''_l(\vec{x})$$

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

Lemma 3.4.11 (Separation). *For every Grzegorczyk polynomial $q(\vec{x})$ and integer $r \geq 0$ one can find a Grzegorczyk polynomial $\bar{q}$ over $\vec{x}^r := \{x \mid \nu(q, x) > r\}$ such that*

(a) $q(\vec{x}) \leq \bar{q}(\vec{x}^r) +^{r+1} \max \vec{x}^*$ *for some* $\vec{x}^* \subseteq \mathsf{Var}(q) \setminus \vec{x}^r$,

(b) $\nu(\bar{q}, x) = \nu(q, x)$ *for all* $x \in \vec{x}^r$.

Proof. Induction on $\nu(q)$. In the *base case* $\nu(q) = 0$ we obtain $q(\vec{x}) \leq c_q +^{r+1} \max \mathsf{Var}(q)$ by Domination (3.4.6) and Lemma 3.4.2 (d). In that case $\bar{q} := c_q$ and $\vec{x}^* := \mathsf{Var}(q)$ will do.

Step case $\nu(q) > 0$. If $r \geq \nu(q)$ then $q(\vec{x}) \leq c_q +^{r+1} \max \mathsf{Var}(q)$ by Domination (3.4.6) and Lemma 3.4.2 (d), hence $\bar{q} := c_q$ and $\vec{x}^* := \mathsf{Var}(q)$ will do. If $r = \nu(q) - 1$ then we apply Top Level Separation (3.4.8) to obtain a Grzegorczyk polynomial $q'(\vec{x}^q)$ such that $q(\vec{x}) \leq q'(\vec{x}^q) +^{\nu(q)} \max \vec{x}^*$ for some $\vec{x}^* \subseteq \mathsf{Var}(q) \setminus \vec{x}^q$, and $\nu(q', x) = \nu(q) = r+1$ for $x \in \vec{x}^q$. 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 $r\text{-}\mathsf{dcp}(q) = q_0(\vec{x}\vec{w}), q_1(\vec{x}), \ldots, q_k(\vec{x})$ of q, satisfying $(i), \ldots, (v)$. As $\nu(q_0) \leq r+1 < \nu(q)$, the I.H. on q_0 yields a Grzegorczyk polynomial $\bar{q}_0$ over $\vec{x}\vec{w}^r = \{u \mid \nu(q_0, u) = r+1\}$ satisfying

(1) $q_0(\vec{x}\vec{w}) \leq \bar{q}_0(\vec{x}\vec{w}^r) +^{r+1} \max \vec{x}\vec{w}^*$ for some $\vec{x}\vec{w}^* \subseteq \mathsf{Var}(q_0) \setminus \vec{x}\vec{w}^r$,

(2) $\nu(\bar{q}_0, u) = \nu(q_0, u) = r+1$ for all $u \in \vec{x}\vec{w}^r$.

Let $\vec{x}_j$ be $\mathsf{Var}(q_j)$ for $j = 1, \ldots, k$. Then we define $\bar{q}$ by

$$\bar{q} := \bar{q}_0((\vec{x}, q_1(\vec{x}_1), \ldots, q_k(\vec{x}_k))^r) +^{r+1} \max(q_1(\vec{x}_1), \ldots, q_k(\vec{x}_k))$$

where it is understood that $q_{j+1}(\vec{x}_{j+1})$ is only substituted for w_{j+1} in $\bar{q}_0(\vec{x}\vec{w}^r)$ if w_{j+1} is in $\vec{x}\vec{w}^r$. Furthermore, let $\vec{x}^*$ be the x_i's in $\vec{x}\vec{w}^*$. Then we obtain:

$$\begin{aligned} q(\vec{x}) &= q_0(\vec{x}, q_1(\vec{x}_1), \ldots, q_k(\vec{x}_k)) && \text{by } (i) \\ &\leq \bar{q}_0((\vec{x}, q_1(\vec{x}_1), \ldots, q_k(\vec{x}_k))^r) +^{r+1} \max((\vec{x}, q_1(\vec{x}_1), \ldots, q_k(\vec{x}_k))^*) && \text{by (1)} \\ &= \bar{q} +^{r+1} \max \vec{x}^* && \text{by (3.4.2) (a), (e)} \end{aligned}$$

Thus, to finish this case, it remains to show $\vec{x}^r = \mathsf{Var}(\bar{q})$ and part (b). First consider any $x \in \vec{x}^r$, that is, $\nu(q, x) \geq r+1$. If $x \in \bigcup \mathsf{Var}(q_{j+1})$ then $\nu(q, x) = \max(\nu(q_1, x), \ldots, \nu(q_k, x))$ by (iii), and that value is $\nu(\bar{q}, x)$; because by (2) every $u \in \mathsf{Var}(\bar{q}_0)$ has modified measure $r+1$, and by (iv) every $u \in \mathsf{Var}(q_{j+1})$ has a modified measure $\geq r+1$. Otherwise if $x \notin \bigcup \mathsf{Var}(q_{j+1})$ then $r+1 \leq \nu(q, x) = \nu(q_0, x)$ by Composition (3.4.9) and (i). Thus we conclude from $(v), (2)$ that $r+1 = \nu(q_0, x) = \nu(\bar{q}_0, x) = \nu(\bar{q}, x)$. In particular, this shows $\vec{x}^r \subseteq \mathsf{Var}(\bar{q})$; the converse holds by a similar argument. □

Lemma 3.4.12 (Modified Bounding). *For every term $t \in \mathsf{PR}_i$ and every list $\vec{x} := x_1, \ldots, x_p$ of distinct variables such that $\vec{x} \supseteq \mathsf{FV}(t)$ and $\mu(\lambda\vec{x}.t) = (\tau_1, \ldots, \tau_{m_t} \to \iota)$, one can find a Grzegorczyk polynomial $q_t(u_1, \ldots, u_{m_t})$ satisfying*

(a) $\quad [\![\lambda\vec{x}.t]\!](\vec{a}, \vec{b}) \leq q_t(\vec{a}, \vec{b}) \quad$ *for* $t \in \mathsf{PR}_1$
$\quad\quad |[\![\lambda\vec{x}.t]\!](\vec{a}, \vec{b})| \leq q_t(|\vec{a}, \vec{b}|) \quad$ *for* $t \in \mathsf{PR}_2$

(b) $\nu(q_t, u_i) = \mathrm{R}(\tau_i)$ *for* $1 \leq i \leq m_t$

(c) $u_i \in \mathsf{Var}(q_t) \Leftrightarrow \tau_i \neq \iota$ *for* $1 \leq i \leq m_t$.

Proof. We write $\mu(\lambda\vec{x}.t) = (\tau_1^t, \ldots, \tau_{m_t}^t \to \iota)$ to explicitly indicate the reference to the term t under consideration. Furthermore, it is understood that the numbering $u_1, \ldots, u_{m_t}$ 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 $\tau_1^t, \ldots, \tau_p^t$, and $u_{p+1}, \ldots, u_{m_t}$ to the (already existing) "input positions" of t, that is, to $\tau_{p+1}^t, \ldots, \tau_{m_t}^t$.

We prove the Modified Bounding lemma for PR_1, and discuss the minor changes for PR_2 afterwards – indeed the proof for PR_2 is practically the same. We proceed by induction on the structure of $t \in \mathsf{PR}_1$, in each case assuming an arbitrary but fixed list $\vec{x} := x_1, \ldots, x_p$ of distinct variables such that $\vec{x} \supseteq \mathsf{FV}(t)$.

If t is a variable x_i, then we set $q_t := u_i$. If t is the constant $\mathbf{0}$ then $q_t := 0$ will do, while if t is the constant $\mathbf{S}$ then we set $q_t := 1 +^1 u_{m_t}$. If t is $\lambda x.r$ 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 is rs where $\mu(r) = X; (\tau_{p+1}^r, \tau_{p+2}^r, \ldots, \tau_{m_r}^r \to \iota)$, 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:

$$[\![\lambda\vec{x}.t]\!](\vec{a}, \vec{b}) \;=\; [\![\lambda\vec{x}.r]\!](\vec{a}, [\![\lambda\vec{x}.s]\!](\vec{a}), \vec{b}) \leq q_r(\vec{a}, q_s(\vec{a}), \vec{b}) \tag{1}$$

If $\tau_{p+1}^r = \iota$ 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 $\mu(\lambda\vec{x}.t) = (\tau_1^r, \ldots, \tau_p^r, \tau_{p+2}^r, \ldots, \tau_{m_r}^r \to \iota)$. So consider the case where $\tau_{p+1}^r = l$. Hence $\mu(rs) = X \cup Y_l, (\tau_{p+2}^r, \ldots, \tau_{m_r}^r \to \iota)$. 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 $\nu(q_t, u_i) = \nu(q_r, u_i) = \mathrm{R}(\tau_i^r) = \mathrm{R}(\tau_i^t)$. As for $1 \leq i \leq p$, first note that

$$\begin{aligned} \mathrm{R}(\tau_i^t) \;&=\; \max\{\mathrm{RANK}(x_i, X), \mathrm{RANK}(x_i, Y_{\restriction l})\} \\ &=\; \max(\{\mathrm{RANK}(x_i, X)\} \cup \{l, \mathrm{RANK}(x_i, Y) \mid x_i \in \mathrm{VAR}(Y)\}) \end{aligned}$$

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

$$\nu(q_t, u_i) = \max(\{\nu(q_r, u_i)\} \cup \{l, \nu(q_s, u_i) \mid u_i \in \mathsf{Var}(q_s)\}).$$

The induction hypothesis on r, s gives $u_i \in \mathsf{Var}(q_s) \Leftrightarrow \tau_i^s \neq \iota \Leftrightarrow x_i \in \mathrm{VAR}(Y)$, and $\nu(q_r, u_i) = \mathrm{R}(\tau_i^r) = \mathrm{RANK}(x_i, X)$ and $\nu(q_s, u_i) = \mathrm{R}(\tau_i^s) = \mathrm{RANK}(x_i, Y)$. Thus, $\nu(q_t, u_i) = \mathrm{R}(\tau_i^t)$.

As to (c), for $p+1 < i \leq m_r$ the induction hypothesis on r yields $u_i \in \mathsf{Var}(q_t) \Leftrightarrow u_i \in \mathsf{Var}(q_r) \Leftrightarrow \iota \neq \tau^r_i = \tau^t_i$. For $1 \leq i \leq p$ the induction hypothesis on r, s together with $u_{p+1} \in \mathsf{Var}(q_r)$ and $\tau^r_{p+1} \neq \iota$ gives $u_i \in \mathsf{Var}(q_t) \Leftrightarrow u_i \in \mathsf{Var}(q_r) \cup \mathsf{Var}(q_s) \Leftrightarrow \tau^r_i \neq \iota \vee \tau^s_i \neq \iota \Leftrightarrow \tau^t_i \neq \iota$. This concludes the application case.

Case t is a recursion $\mathbf{R}\, r\, s$. 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 $\mu(r) = X; \iota$ and $\mu(s) = Y; (\tau^s_{p+1}, \tau^s_{m_s} \to \iota)$. 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 $\tau^s_{m_s} = \iota$. Hence $\mu(t) = X \cup Y; (\mathrm{R}(\tau^s_{p+1}) \to \iota)$ and $u_{m_s} \notin \mathsf{Var}(q_s)$ 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 $\tau^s_{m_s} = l$. Hence $\nu(q_s, u_{m_s}) = l$ by the induction hypothesis on s, and furthermore,

$$\mu(t) = X_l \cup Y_l; (k \to \iota) \tag{2}$$

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

$$q_s(\vec{u}) \leq \bar{q}_s(\vec{u}^l) +^{l+1} \max(\vec{u}^*, u_{m_s}) \tag{3}$$

$$\nu(\bar{q}_s, u) = \nu(q_s, u) \text{ for every } u \in \vec{u}^l \tag{4}$$

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

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

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 $f(\vec{a}, 0) \leq q_r(\vec{a}) \leq q_t(\vec{a}, 0)$ by the induction hypothesis on r and Lemma 3.4.2. In the *step case* $b \to b+1$, we conclude (a) as follows:

$$\begin{aligned}
f(\vec{a}, b+1) &\leq q_s(\vec{a}, b, f(\vec{a}, b)) && \text{by def. and the I.H. on } s \\
&\leq \bar{q}_s(\vec{a}b^l) +^{l+1} \max(\vec{a}b^*, q_t(\vec{a}, b)) && \text{by (3), the I.H. on } b, \text{(3.4.2)(d)} \\
&= S^{l+1(\bar{q}_s(\vec{a}b^l))}(q_t(\vec{a}, b)) && \text{by (3.4.2) (a), (d)} \\
&= S^{l+1(\bar{q}_s(\vec{a}b^l))}(S^{l+1(b\times^1 \bar{q}_s(\vec{a}b^l))}(\max(\vec{a}b^*, q_r(\vec{a})))) && \text{by def. of } q_t \text{ and (3.4.2) (a)} \\
&= S^{l+1((b+1)\times^1 \bar{q}_s(\vec{a}b^l))}(\max(\vec{a}b^*, q_r(\vec{a})))) && \text{by } \times^1 = \times \\
&\leq q_t(\vec{a}, b+1) && \text{by (3.4.2) (d), (a) two times}
\end{aligned}$$

As for (b), if $u_{p+1} \in \vec{u}^l$ then $\nu(\bar{q}_s, u_{p+1}) = \nu(q_s, u_{p+1}) \geq l+1$ by (4). Thus, (2) and the induction hypothesis on q_s imply $\mathrm{R}(\tau^t_{m_t}) = k = \mathrm{R}(\tau^s_{p+1}) = \nu(q_s, u_{p+1}) = \nu(\bar{q}_s, u_{p+1}) = \nu(q_t, u_{p+1})$. Otherwise if $u_{p+1} \notin \vec{u}^l$, then (2) and the induction hypothesis on q_s imply $\mathrm{R}(\tau^t_{m_t}) = k = l+1 = \nu(q_t, u_{p+1})$.

As for a parameter u_i ($1 \leq i \leq p$), first consider the case $u_i \in \vec{u}^l$. Hence $\nu(\bar{q}_s, u_i) = \nu(q_s, u_i) \geq l+1$ by (4). Therefore the induction hypothesis implies $\nu(q_t, u_i) = \max\{\nu(q_s, u_i), \nu(q_r, u_i)\} = \max\{\mathrm{R}(\tau^s_i), \mathrm{R}(\tau^r_i)\} = \max\{\mathrm{R}(\tau^s_i), \mathrm{R}(\tau^r_i), l\} = \mathrm{R}(\tau^t_i)$. Otherwise if $u_i \notin \vec{u}^l$, then $\nu(q_s, u_i) = \mathrm{R}(\tau^s_i) \leq l$ by the induction hypothesis on s. If $u_i \in \vec{u}^*$, then the I.H. on q_r gives $\nu(q_t, u_i) = \max\{l, \nu(q_r, u_i)\} = \max\{l, \mathrm{R}(\tau^r_i)\}$, and this value is also $\mathrm{R}(\tau^t_i)$. Otherwise if $u_i \notin \vec{u}^*$, then $u_i \notin \mathsf{Var}(q_s)$, hence $\tau^s_i = \iota$ by the induction hypothesis on

s. In that case the definition of modified measure ν and the induction hypothesis on q_r imply $\nu(q_t, u_i) = \max\{l, \nu(q_r, u_i) \mid u_i \in \mathsf{Var}(q_r)\} = \max\{l, \mathrm{R}(\tau_i^r) \mid \tau_i^r \neq \iota\}$. This value is also $\mathrm{R}(\tau_i^t)$, concluding the proof of (b).

Concerning (c), first note that $u_{p+1} \in \mathsf{Var}(q_t)$ and $\tau_{m_t}^t = k$. As for a parameter u_i, first observe that $u_i \in \mathsf{Var}(q_s) \Leftrightarrow u_i \in \mathsf{Var}(\bar{q}_s) \vee u_i \in \vec{u}^*$. Thus we conclude from the induction hypothesis on r, s that $u_i \in \mathsf{Var}(q_t) \Leftrightarrow u_i \in \mathsf{Var}(q_s) \vee u_i \in \mathsf{Var}(q_r) \Leftrightarrow \mathrm{R}(\tau_i^s) \neq \iota \vee \mathrm{R}(\tau_i^r) \neq \iota \Leftrightarrow \mathrm{R}(\tau_i^t) \neq \iota$. 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 is $\mathbf{RN}\, r\, t_0\, t_1$. 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 $q_{t_i}(u_1, \ldots, u_p, u_{p+1}, u_{p+2})$. 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 ν for Grzegorczyk polynomials $\max(q_1, q_2)$, all arguments for q_s carry over to q. More precisely, if $\tau_{m_{p+2}}^{t_0} = \iota$ and $\tau_{m_{p+2}}^{t_1} = \iota$, we define

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

Otherwise if $\tau_{m_{p+2}}^{t_0} \neq \iota$ or $\tau_{m_{p+2}}^{t_1} \neq \iota$, Separation (3.4.11) for q and $l := \max\{\mathrm{R}(\tau_{m_{p+2}}^{t_0}), \mathrm{R}(\tau_{m_{p+2}}^{t_1})\}$ yields a Grzegorczyk polynomial $\bar{q}$ satisfying (3),(4) above. Now let

$$q_t(u_1, \ldots, u_p, u_{p+1}) := (u_{p+1} \times^1 \bar{q}(\vec{u}^l)) +^{l+1} \max(\vec{u}^*, q_r(u_1, \ldots, u_p)).$$

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

Theorem 3.4.13 (Bounding). *For every closed term $t \in \mathsf{PR}_i$ of type $\vec{\iota} \to \iota$ one can find a Grzegorczyk polynomial $q_t \in \mathcal{E}_{\mu(t)+1}$ satisfying $[\![t]\!](\vec{x}) \leq q_t(\vec{x})$ for $t \in \mathsf{PR}_1$, and $|[\![t]\!](\vec{x})| \leq q_t(|\vec{x}|)$ for $t \in \mathsf{PR}_2$.*

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

3.5. Characterisation Theorems

The measure μ on 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 *nth modified Heinermann class* $\mathcal{R}_i^n$ is defined by

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

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 $\mathcal{R}_1^1$ and a "witness" w_f in $\mathcal{R}_1^n$ 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 $\mathcal{R}_1^n$. The method generalises Bellantoni's proof [3] that the functions in $\mathcal{E}_2$ can be simulated by safe primitive recursion. Since $\mathcal{R}_1^1$ is closed under composition, this improves the result in [3]. The inclusion $\supseteq$ follows from the Bounding Theorem by a straightforward inductive argument, because Bounding (3.4.13) turns every primitive recursion in $\mathcal{R}_1^n$ into a bounded primitive recursion in $\mathcal{E}_{n+1}$.

In the sequel we sometimes mix closed terms $f \in \mathsf{PR}_i$ with the functions $[\![f]\!]$ denoted.

Lemma 3.5.2 ($\mathcal{E}$-Bounding). *Every $f \in \mathcal{E}_{n+2}$ has a monotone increasing bound $b_f \in \mathcal{R}_1^{n+1}$.*

Proof. First recall that $A_2 \equiv \times^1$ and $\mu(\times^1) = (1, 1 \to \iota)$. Under this definition of A_2, A_{m+3} is defined inductively by a top recursion from A_{m+2}, that is, $A_{m+3} := \lambda xy.\mathbf{R}\,\mathbf{S0}\,(\lambda uv.A_{m+2}xv)\,y$. Thus, we obtain

$$\mu(A_{m+3}) = (m+1, m+2 \to \iota).$$

Note that $\max$ is in $\mathcal{R}_1^1$, as $\max = \lambda xy.\mathsf{C}\,(\dot{-}\ xy)\,y\,x$ where $\lambda xy.\mathbf{R}\,x\,(\lambda uv.\mathsf{P}v)\,y$ defines *modified subtraction* $\dot{-}$, and C, P are as in note 1. Recall that every f in $\mathcal{E}_{n+2}$ 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, $A_{n+3}(\max(2,\vec{x}), c_f) = A_{n+2}(\max(2,\vec{x}), \ldots, A_{n+2}(\max(2,\vec{x}), 1)\ldots)$ ($c_f - 1$ calls of A_{n+2}), we obtain a monotone increasing bound on f in $\mathcal{R}_1^{n+1}$. 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). $\forall n \geq 1.\ \mathcal{E}_{n+1} = \mathcal{R}_1^n$

Proof. The inclusion $\subseteq$ follows from (A) below, for given a function $f \in \mathcal{E}_{n+1}$, $n \geq 1$, then (A) implies $f(\vec{x}) = f^*(w_f(\vec{x}), \vec{x})$ for some $f^* \in \mathcal{R}_1^1$ and $w_f \in \mathcal{R}_1^n$.

(A) For every $f \in \mathcal{E}_{\ell+2}$ one can find a *simulation* $f^* \in \mathsf{PR}_1$ and a monotone increasing *witness* $w_f \in \mathcal{R}_1^{\ell+1}$ such that $f^*(w,\vec{x}) = f(\vec{x})$ for $w \geq w_f(\vec{x})$, and $\mu(f^*) = (\tau_0, \tau_1, \ldots, \tau_{m_f} \to \iota)$ with $\mathrm{R}(\tau_0) \leq 1$, and $\mathrm{R}(\tau_i) = 0$ for $i = 1, \ldots, m_f$.

The proof of (A) is by induction on the definition of $f \in \mathcal{E}_{\ell+2}$. If f is one of the initial functions $0, S, \Pi_j^m$ then $f^* := \lambda w.\mathbf{0}, \lambda wx.\mathbf{S}x, \lambda w\vec{x}.x_i$ respectively and $w_f := \lambda\vec{x}.\mathbf{0}$ 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 $\mathcal{E}_{\ell+2}$ 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 $\mathcal{R}_1^{\ell+1}$. By $\mathcal{E}$-Bounding (3.5.2) we obtain a monotone increasing bound $b \in \mathcal{R}_1^{\ell+1}$ on all $g_1, \ldots, g_m$. We define f^* and w_f by:

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

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 $g, h, k \in \mathcal{E}_{\ell+2}$. The induction hypothesis yields simulations h^*, g^* in PR_1 with monotone increasing witness functions w_h, w_g in $\mathcal{R}_1^{\ell+1}$. The required witness w_f can be defined by

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

where $b \in \mathcal{R}_1^{\ell+1}$ 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 $\mathcal{R}_1^1$ which *simulates f up to y*, that is,

(i) If $w \geq w_f(\vec{x}, y)$ then $\hat{f}(\hat{w}, w, \vec{x}, y) = \begin{cases} f(\vec{x}, \hat{w}) & \text{if } \hat{w} \leq y \\ f(\vec{x}, y) & \text{else} \end{cases}$

(ii) $\mu(\hat{f}) = (1, \tau_0, \tau_1, \ldots, \tau_{m_f} \to \iota)$ such that $\mathrm{R}(\tau_0) \leq 1$, and $\mathrm{R}(\tau_i) = 0$ for $i = 1, \ldots, m_f$.

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

$$\begin{aligned} \hat{f}(0, w, \vec{x}, y) &:= g^*(w, \vec{x}) \\ \hat{f}(\hat{w}{+}1, w, \vec{x}, y) &:= \begin{cases} h^*(w, \vec{x}, \hat{w}, \hat{f}(\hat{w}, w, \vec{x}, y)) & \text{if } \hat{w} + 1 \leq y \\ \hat{f}(\hat{w}, w, \vec{x}, y) & \text{else} \end{cases} \end{aligned}$$

To prove (i), it suffices to show that $w \geq w_f(\vec{x}, y)$ implies $\hat{f}(\hat{w}, w, \vec{x}, y) = f(\vec{x}, \hat{w})$ for $\hat{w} \leq y$. So assume that $w \geq w_f(\vec{x}, y)$, and we proceed by induction on $\hat{w} \leq y$. First observe that by monotonicity we obtain (*) $w \geq w_h(\vec{x}, y, b(\vec{x}, y)) + w_g(\vec{x}) + y \geq w_h(\vec{x}, \hat{w}, f(\vec{x}, \hat{w})) + w_g(\vec{x}) + y$. The base case $\hat{w} = 0$ follows from the induction hypothesis on g and (*), since $\hat{f}(0, w, \vec{x}, y) = g^*(w, \vec{x}) = g(\vec{x}) = f(\vec{x}, 0)$. The *step case* $\hat{w} \to \hat{w} + 1 \leq y$ is concluded as follows:

$$\begin{aligned} \hat{f}(\hat{w}+1, w, \vec{x}, y) &= h^*(w, \vec{x}, \hat{w}, \hat{f}(\hat{w}, w, \vec{x}, y)) && \text{by definition} \\ &= h^*(w, \vec{x}, \hat{w}, f(\vec{x}, \hat{w})) && \text{by the I.H. on } \hat{w} \\ &= h(\vec{x}, \hat{w}, f(\vec{x}, \hat{w})) && \text{by the I.H. on } h \text{ and (*)} \\ &= f(\vec{x}, \hat{w}+1) && \text{by definition} \end{aligned}$$

To achieve (ii), we need to define the characteristic function of $\leq$ such that $\mu(c_{\leq}) = (1, 0 \to \iota)$. Since $x \leq y \Leftrightarrow y{+}1 \dot{-} x > 0$, we can do this by setting $c_{\leq} := \lambda x y.\mathsf{C}\,(\dot{-}\,(\mathbf{S}y)x)\,\mathbf{0}\,(\mathbf{S0})$, where C is defined as in note 1, and $\mu(\dot{-}) = (0, 1 \to \iota)$. Thus, following the recursion equations of $\hat{f}$, we define:

$$\hat{f} := \lambda \hat{w} w \vec{x} y.\mathbf{R}\,(g^* w \vec{x})\,(\lambda u v.\mathsf{C}\,(c_{\leq}(\mathbf{S}u)y)\,v\,(h^* w \vec{x} u v))\,\hat{w}$$

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

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

(B) For every $t \in \mathsf{PR}_1$ and every $\vec{x} \supseteq \mathsf{FV}(t)$, $[\![\lambda \vec{x}.t]\!]$ 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 $\mathbf{0}, \mathbf{S}$, we have $[\![\lambda \vec{x}.t]\!] \in \mathcal{E}_0$. The case where t is $\lambda x.r$ follows from the induction hypothesis on r with respect to $\vec{x}, x$. The case where t is rs follows from the induction hypothesis on r and s, because $[\![\lambda \vec{x}.t]\!](\vec{a}, \vec{b}) = [\![\lambda \vec{x}.r]\!](\vec{a}, [\![\lambda \vec{x}.s]\!](\vec{a}), \vec{b})$ and fairness of t implies $\mu(t) = \max(\mu(r), \mu(s))$. So consider the case where t is $\mathbf{R}\,r\,s$. The induction hypothesis on r and s implies $[\![\lambda \vec{x}.r]\!], [\![\lambda \vec{x}.s]\!] \in \mathcal{E}_{\mu(t)+1}$, and Bounding (3.4.13) gives that $[\![\lambda \vec{x}.t]\!]$ has a bound in $\mathcal{E}_{\mu(t)+1}$. Thus, $[\![\lambda \vec{x}.t]\!]$ is defined in $\mathcal{E}_{\mu(t)+1}$. □

A further benefit of the Bounding Theorem, together with the ability of μ 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 FPTIME (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 $\mathsf{shift\text{-}left}$ satisfying $\mathsf{shift\text{-}left}(x,y) = s_0^{(|x|)}(y) = 2^{|x|}\cdot y$ can be defined by

$$\mathsf{shift\text{-}left} := \lambda xy.\, \mathbf{RN}\, y\, (\lambda uv.\mathbf{s}_0 v)\, (\lambda uv.\mathbf{s}_0 v)\, x$$

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

$$\mathrm{smash} := \lambda xy.\, \mathbf{RN}\, \mathbf{s}_1\mathbf{0}\, (\lambda uv.\mathsf{shift\text{-}left}\, yv)\, (\lambda uv.\mathsf{shift\text{-}left}\, yv)\, x$$

Accordingly, we obtain $\mu(\mathrm{smash}) = (1, 1 \to \iota)$ and hence $\mathrm{smash} \in \mathcal{R}_2^1$ as required. □

Theorem 3.5.5 (Polynomial Time). $\mathcal{R}_2^1 = \mathrm{FPTIME}$

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 $t \in \mathsf{PR}_2$ with $\mu(t) \leq 1$, and $\vec{x} \supseteq \mathsf{FV}(t)$, $[\![\lambda\vec{x}.t]\!]$ belongs to FP.

The inclusion $\supseteq$ follows from (D) below, since each $f \in \mathrm{FP}$ 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 $\lambda\vec{x}.f^*(2^{p_f(|\vec{x}|)})\vec{x}$ defines f in $\mathcal{R}_2^1$.

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

The proof is by induction on the definition of $f \in \mathrm{FP}$. 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 $f^* := \lambda w\vec{x}.f\vec{x}$ and $p_f := \lambda\vec{x}.\mathbf{0}$. 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 $\mathsf{shift\text{-}left}$ by bounded recursion on notation in FP, using smash itself as bounding function. Observe that $\mathsf{shift\text{-}left}$ is defined in FP by bounded recursion on notation, since $\mathsf{shift\text{-}left}(x,y) = 2^{|x|}\cdot y \leq 2^{|x|+|y|} \leq \mathrm{smash}(s_1(x), s_1(y))$.

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

$$f^* := \lambda w\vec{x}.h^* w(g_1^* w\vec{x}) \ldots (g_m^* w\vec{x})$$
$$p_f(\vec{x}) := p_h(q(\vec{x}), \ldots, q(\vec{x})) + \sum_j p_{g_j}(\vec{x}),$$

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 $g,h,k \in \mathsf{FP}$. The induction hypothesis yields simulations $g^*, h_0^*, h_1^* \in \mathsf{PR}_2$ 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^* of f, we first define a function $\hat{f}$ in $\mathcal{R}_2^1$ such that $\mu(\hat{f}) = (1,1,\tau_1,\ldots,\tau_{m_f} \to \iota)$ with $\tau_1,\ldots,\tau_{m_f} \in \{\iota, 0\}$, and $\hat{f}$ *simulates* f as follows: For $|w| \geq p_f(|\vec{x},y|)$,

$$(*) \qquad \text{if } |w|-|y| \leq |\hat{w}| \leq |w|, \text{ then } \hat{f}(\hat{w},w,\vec{x},y) = f(p^{(|w|-|\hat{w}|)}(y),\vec{x}).$$

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

First we define the function Y satisfying $Y(\hat{w},w,y) = p^{(|w| \dot{-} |\hat{w}|)}(y)$ and $\mu(Y) = (1,1,0 \to \iota)$. That is obtained by $Y := \lambda\hat{w}wy.\ominus(\ominus\hat{w}w)y$, where $\ominus$ satisfying $x \ominus y = p^{(|x|)}(y)$ and $\mu(\ominus) = (1,0 \to \iota)$ is defined from p (cf. Note 1) by $\ominus := \lambda xy.\mathbf{RN}\, y\, (\lambda uv.\mathsf{p}v)(\lambda uv.\mathsf{p}v)\, x$. Observe that $|x \ominus y| = |y| \dot{-} |x|$.

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

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

$$\hat{f}(s_i(\hat{w}),w,y,\vec{x}) = \begin{cases} g^*(w,\vec{x}) & |s_i(\hat{w})| \leq |w|-|y| \\ h_0^*(w,Y(\hat{w},w,y),\vec{x},\hat{f}(\hat{w},w,y,\vec{x})) & |s_i(\hat{w})| > |w|-|y| \text{ and } Y(s_i(\hat{w}),w,y) \text{ is even} \\ h_1^*(w,Y(\hat{w},w,y),\vec{x},\hat{f}(\hat{w},w,y,\vec{x})) & |s_i(\hat{w})| > |w|-|y| \text{ and } Y(s_i(\hat{w}),w,y) \text{ is odd} \end{cases}$$

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 $\hat{f}(\hat{w},w,\vec{x},y) = g^*(w,\vec{x}) = g(\vec{x}) = f(0,\vec{x})$. 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 $|w| \geq p_{h_0}(|y,\vec{x}|,q_f(|y,\vec{x}|)) \geq p_{h_0}(|Y(\hat{w},w,y),\vec{x}|,|f(Y(\hat{w},w,y),\vec{x})|)$, the I.H. on $\hat{w}$ and that on h imply $\hat{f}(s_i(\hat{w}),w,\vec{x},y) = h_0^*(w,Y(\hat{w},w,y),\vec{x},f(Y(\hat{w},w,y),\vec{x})) = f(Y(s_i(\hat{w}),w,y),\vec{x})$.

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 $\hat{f}$ can be realised as $\hat{f} := \lambda\hat{w}wy\vec{x}.\mathbf{RN}\,\mathbf{0}\, H_0\, H_1\, \hat{w}$, where H_i is defined from binary cases c (cf. Note 1), satisfying $\mu(c) = (0,0,0,0 \to \iota)$, as follows:

$$H_i := \lambda uv.c\,(Y(\mathbf{s}_i u)wy)\,(g^* w\vec{x})\,(h_0^* w(Yuwy)\vec{x}v))\,(h_1^* w(Yuwy)\vec{x}v))$$

□

Theorem 3.5.6 (At and above Kalmár-elementary). $\forall n \geq 2.\ \mathcal{R}_2^n = \mathcal{R}_1^n$

Proof. As to the inclusion $\supseteq$, first recall (cf. Background 2.4) the definition of the unary function $\odot$ used to synchronise the number of steps in a recursion on notation on $\odot(x)$ with the number of steps in a primitive recursion on x, that is, $|\odot(x)| = x$. This function is defined in $\mathcal{R}_2^2$ by a top recursion from concatenation $\oplus$:

$$\odot := \lambda x.\mathbf{RN}\,\mathbf{0}\,(\lambda yz.\oplus zz)\,(\lambda yz.\mathbf{s}_1(\oplus zz))\,x.$$

Recall that $x \oplus y = x \cdot 2^{|y|} + y$. This implies $\odot(x) = 2^x - 1$, and hence $|\odot(x)| = x$ and $p(\odot(x)) = \odot(x \dot{-} 1)$. Now the inclusion $\supseteq$ follows from (E) below, for given a closed term $t \in \mathsf{PR}_1$, (E) yields a closed term $t^* \in \mathsf{PR}_2$ satisfying $|[\![t^*]\!](\odot(\vec{b}))| = [\![t]\!](\vec{b})$, and t, t^* have identical μ-measures. As $|\cdot|$ belongs to $\mathcal{R}_2^1$ by theorem 3.5.5, we conclude $\mathcal{R}_2^n \supseteq \mathcal{R}_1^n$ for $n \geq 2$.

(E) For every $t \in \mathsf{PR}_1$ and $\vec{x} \supseteq \mathsf{FV}(t)$ with $\mu(\lambda\vec{x}.t) = (\tau_1, \ldots, \tau_{m_t} \to \iota)$ one can find a *simulation* $t^* \in \mathsf{PR}_2$ such that $[\![\lambda\vec{x}.t^*]\!](\odot(\vec{a}, \vec{b})) = \odot([\![\lambda\vec{x}.t]\!](\vec{a}, \vec{b}))$, and $\mu(\lambda\vec{x}.t^*) = (\tau_1, \ldots, \tau_{m_t} \to \iota)$.

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

$$\begin{aligned}(\lambda x.r)^* &:= \lambda x.r^* \\ (rs)^* &:= r^* s^* \\ (\mathbf{R}\, r\, s)^* &:= \mathbf{RN}\, r^*\, s^*\, s^*\end{aligned}$$

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

(F) For every $t \in \mathsf{PR}_2$ and $\vec{x} \supseteq \mathsf{FV}(t)$, $[\![\lambda\vec{x}.t]\!]$ 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 $t \in \mathsf{PR}_2$. All cases are as in the proof of (B), except the case where t is a recursion $\mathbf{RN}\, r\, t_0\, t_1$. By the induction hypothesis the functions $g := [\![\lambda\vec{x}.r]\!], h_0 := [\![\lambda\vec{x}.t_0]\!], h_1 := [\![\lambda\vec{x}.t_1]\!]$ all belong to $\mathcal{E}_{\max(3,\mu(t)+1)}$. So $[\![\lambda\vec{x}.t]\!]$ represents the function f defined by recursion on notation from g, h_0, h_1. Now we reconsider the reduction of recursion on notation to primitive recursion given in Background 2.4. There we defined by one primitive recursion from g, h_0, h_1 and auxiliary functions (all in $\mathcal{E}_1$) a function $\hat{f}$ satisfying

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

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 $|p^{(|y| \dot{-} z)}(y)| \leq |y|$ we obtain

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

This gives the required bound in $\mathcal{E}_{\max(3,\mu(t)+1)}$. □

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

4.1. Stack programs

We presuppose a fixed, but arbitrary alphabet $\Sigma := \{\mathsf{a}_1, \ldots, \mathsf{a}_l\}$. We will define a stack programming language over Σ where programs are built from primitive instructions `push(a,X)`, `pop(X)`, `nil(X)` by sequencing, conditional statements and loop statements. We assume an infinite supply of variables X, Y, Z, 0, U, V, possibly with subscripts. Intuitively, variables serve as *stacks*, each holding an arbitrary word over Σ 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* `push(a,X)`, `pop(X)`, `nil(X)` is a stack program.
- If $\mathsf{P}_1, \mathsf{P}_2$ 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 `if top(X)`$\equiv$`a [P]`.
- If P is a stack program with no occurrence of `push(a,X)`, `pop(X)` or `nil(X)`, then so is the *loop* statement `foreach X [P]`.

We use $\mathcal{V}(\mathsf{P})$ 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 $\{A\}\,\mathsf{P}\,\{B\}$, 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,

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

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

$$\{\vec{\mathsf{X}} = \vec{w}\}\,\mathsf{P}\,\{|\mathsf{X}_1| \leq f_1(|\vec{w}|), \ldots, |\mathsf{X}_n| \leq f_n(|\vec{w}|)\}$$

meaning that *if the words $\vec{w}$ are stored in stacks $\vec{\mathsf{X}}$, 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 $\mathtt{P}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{P}_k$ are executed one by one from the left to the right, where

- the operational semantics of imperatives and conditionals is as expected:
  - `push(a,X)` pushes letter `a` on top of stack `X`,
  - `pop(X)` removes the top symbol on stack `X`, if any, otherwise (`X` is empty) the statement is ignored,
  - `nil(X)` empties stack `X`,
  - `if top(X)≡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 top(X)≡a [Q]`) with `U`; then one executes the sequence

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

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 $f\colon (\Sigma^*)^n \to \Sigma^*$ if `P` has an *output variable* `O` and *input variables* $\mathtt{X}_{i_1}, \ldots, \mathtt{X}_{i_l}$ among stacks $\mathtt{X}_1, \ldots, \mathtt{X}_m$ such that for all $w_1, \ldots, w_n \in \Sigma^*$,

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

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

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 μ 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 $\mathcal{U}(\mathtt{P})$ and $\mathcal{C}(\mathtt{P})$.

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

Definition 4.2.1 (Control). Let `P` be a stack program. The relation *control in P*, denoted $\overset{\mathtt{P}}{\to}$, is defined as the transitive closure of the following binary relation $\prec_{\mathtt{P}}$ on $\mathcal{V}(\mathtt{P})$:

$$\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y} :\Leftrightarrow \mathtt{P} \text{ contains a loop } \mathtt{foreach\ X\ [Q]} \text{ such that } \mathtt{Y} \in \mathcal{U}(\mathtt{Q}).$$

We say that `X` *controls* `Y` *in* `P` if $\mathtt{X} \overset{\mathtt{P}}{\to} \mathtt{Y}$, that is, there exist variables $\mathtt{X} \equiv \mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_l \equiv \mathtt{Y}$ such that $\mathtt{X}_1 \prec_{\mathtt{P}} \mathtt{X}_2 \prec_{\mathtt{P}} \ldots \prec_{\mathtt{P}} \mathtt{X}_{l-1} \prec_{\mathtt{P}} \mathtt{X}_l$.

[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 μ-measure). The *μ-measure* of a stack program P, denoted by $\mu(\mathtt{P})$, is inductively defined as follows:

- $\mu(\mathtt{imp}) := 0$ for every imperative imp.
- $\mu(\mathtt{if\ top(X)} \equiv \mathtt{a\ [Q]}) := \mu(\mathtt{Q})$
- $\mu(\mathtt{Q}_1\mathtt{;}\ \mathtt{Q}_2) := \max\{\mu(\mathtt{Q}_1), \mu(\mathtt{Q}_2)\}$
- If P is a loop $\mathtt{foreach\ X\ [Q]}$, then

$$\mu(\mathtt{P}) := \begin{cases} \mu(\mathtt{Q}) + 1 & \text{if Q is a sequence with a top circle} \\ \mu(\mathtt{Q}) & \text{else} \end{cases}$$

  where $\mathtt{Q} \equiv \mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_l$ has a *top circle* if there exists a component $\mathtt{Q}_i$ with $\mu(\mathtt{Q}_i) = \mu(\mathtt{Q})$ such that some Y controls some Z in $\mathtt{Q}_i$, and Z controls Y in $\mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_{i-1}\mathtt{;}\ \mathtt{Q}_{i+1}\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_l$.

We say that a stack program P *has μ-measure n* if $\mu(\mathtt{P}) = n$.

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 μ-measure n has a *length bound* $b \in \mathcal{E}^{n+2}$, that is, $|f(\vec{w})| \leq b(|\vec{w}|)$ for all $\vec{w}$. 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 μ-measure 0 compute only polynomially length-bounded functions. For the general case, we show that every core program P of μ-measure $n+1$ has a "length bound" $\mathtt{P}'$ of μ-measure $n+1$. The structure of $\mathtt{P}'$ we call *flattened out* will be such that by a straightforward inductive argument we obtain that every function computed by $\mathtt{P}'$ has a length bound in $\mathcal{E}^{n+2}$.

Definition 4.3.1 (Core programs). *Core programs* are stack programs built from imperatives 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* $\vec{\mathtt{X}}$*, then*

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

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

Lemma 4.3.3 (Measure Zero). *For core programs* $\mathtt{P} :\equiv \mathtt{foreach\ X\ [Q]}$ *with* $\mu(\mathtt{P}) = 0$, $\xrightarrow{\mathtt{P}}$ *is irreflexive.*

Proof. By induction on the structure of core programs $\mathtt{P} :\equiv \mathtt{foreach\ X\ [Q]}$ of μ-measure 0. The statement is obvious if Q is an imperative push(a,Y). If Q is of the form $\mathtt{foreach\ Y\ [R]}$, the statement follows from the induction hypothesis on Q and $\mathtt{X} \notin \mathcal{U}(\mathtt{Q})$. Finally, if Q is a sequence $\mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_n$, then by the induction hypothesis on each component $\mathtt{Q}_i$, no Y controls Y in $\mathtt{Q}_i$. Therefore, if some Y controlled Y in Q, then Y controlled some $\mathtt{Z} \not\equiv \mathtt{Y}$ in some $\mathtt{Q}_j$, and Z controlled Y in the context $\mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_{j-1}\mathtt{;}\ \mathtt{Q}_{j+1}\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_n$. Hence Q would have a top circle, contradicting the hypothesis $\mu(\mathtt{P}) = 0$. □

Lemma 4.3.4 (Irreflexive Bounding). *Let* P *be a core program with irreflexive* $\xrightarrow{\mathsf{P}}$*. Let* P *have variables among* $\vec{\mathsf{X}} := \mathsf{X}_1, \ldots, \mathsf{X}_n$*, and for* $i = 1, \ldots, n$ *let* V^i *denote the list of those variables* X_j *which control* X_i *in* P*. Then there are polynomials* $p_1(V^1), \ldots, p_n(V^n)$ *such that for all* $\vec{w} := w_1, \ldots, w_n$*,*

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

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 P with irreflexive $\xrightarrow{\mathsf{P}}$. In the *base case* where P is of the form $\mathtt{push(a,X_1)}$, we know $V^1 = \emptyset$ and hence $p_1 := 1$ will do.

Step case P is a sequence $\mathsf{P}_1\mathtt{;}\ \mathsf{P}_2$. The induction hypothesis yields polynomials $q_1(V^1), \ldots, q_n(V^n)$ for P_1, and polynomials $r_1(V^1), \ldots, r_n(V^n)$ for P_2. Now fix any i among $1, \ldots, n$, and suppose that $\mathsf{X}_{i_1}, \ldots, \mathsf{X}_{i_l}$ are the variables which control X_i in P. Then one easily verifies that

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{P}\ \{|\mathsf{X}_i| \leq |w_i| + q_i(|\vec{w}^i|) + r_i(|w_{i_1}| + q_{i_1}(|\vec{w}^{i_1}|), \ldots, |w_{i_l}| + q_{i_l}(|\vec{w}^{i_l}|))\}.$$

Step case P is a loop $\mathtt{foreach\ X_j\ [Q]}$. The I.H. on Q yields polynomials $p_1(V^1), \ldots, p_n(V^n)$ so that

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{Q}\ \{|\mathsf{X}_i| \leq |w_i| + p_i(|\vec{w}^i|)\} \quad \text{for } i = 1, \ldots, n. \tag{1}$$

As $\xrightarrow{\mathsf{P}}$ is irreflexive, then so is $\xrightarrow{\mathsf{Q}}$. From this it follows that $\xrightarrow{\mathsf{Q}}$ defines a strict partial order on $\vec{\mathsf{X}}$. 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 $m, \vec{w}$,

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{Q}^m\ \{|\mathsf{X}_i| \leq |w_i| + q_i(m, |\vec{w}^i|)\} \quad \text{for } i = 1, \ldots, n \tag{2}$$

where Q^m denotes the sequence $\mathsf{Q}\mathtt{;}\ \ldots\ \mathtt{;}\ \mathsf{Q}$ (m times). Note that (2) implies the statement of the lemma for the current case $\mathsf{P} \equiv \mathtt{foreach\ X_j\ [Q]}$. To see this, if $\mathsf{X}_i \notin \mathcal{U}(\mathsf{Q})$ 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 $\mathsf{X}_i \in \mathcal{U}(\mathsf{Q})$, then $\mathsf{X}_j \in V^i$ and (2) gives $\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{P}\ \{|\mathsf{X}_i| \leq |w_i| + q_i(|w_j|, |\vec{w}^i|)\}$ where $w_j \in \vec{w}^i$.

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 $\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{Q}^m\ \{|\mathsf{X}_i| \leq |w_i| + m \cdot p_i\}$. So consider the case $V^i = \mathsf{X}_{i_1}, \ldots, \mathsf{X}_{i_l}$ with $l \geq 1$. The side induction hypothesis yields polynomials $r_{i_1}(m, V^{i_1}), \ldots, r_{i_l}(m, V^{i_l})$ such that for all $m, \vec{w}$,

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{Q}^m\ \{|\mathsf{X}_{i_j}| \leq |w_{i_j}| + r_{i_j}(m, |\vec{w}^{i_j}|)\} \quad \text{for } j = 1, \ldots, l \tag{3}$$

where V^{i_j} denotes the variables which control X_{i_j} in Q. Observe that $\mathsf{X}_{i_j} \notin V^{i_j}$ for $j = 1, \ldots, l$ and $\mathsf{X}_i \notin V^{i_1} \cup \ldots \cup V^{i_l} \subseteq V^i$. Hence it suffices to prove by induction on m that for all $m, \vec{w}$,

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{Q}^m\ \{|\mathsf{X}_i| \leq |w_i| + m \cdot p_i(|w_{i_1}| + r_{i_1}(m, |\vec{w}^{i_1}|), \ldots, |w_{i_l}| + r_{i_l}(m, |\vec{w}^{i_l}|))\}. \tag{4}$$

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

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{Q}^m\ \{\mathsf{X}_{i_1} = u_{i_1}, \ldots, \mathsf{X}_{i_l} = u_{i_l}, \mathsf{X}_i = v_i, \ldots\}$$
$$\{\mathsf{X}_{i_1} = u_{i_1}, \ldots, \mathsf{X}_{i_l} = u_{i_l}, \mathsf{X}_i = v_i, \ldots\}\ \ \mathsf{Q}\ \ \{\mathsf{X}_i = v_i^*\}.$$

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

$$\begin{array}{ll}
|v_i^*| \leq |v_i| + p_i(|u_{i_1}|, \ldots, |u_{i_l}|) & \text{by the main I.H. (1)} \\
|v_i| \leq |w_i| + m \cdot p_i(\ldots, |w_{i_j}| + r_{i_j}(m, |\vec{w}^{i_j}|), \ldots) & \text{by the I.H. for } m \\
|u_{i_j}| \leq |w_{i_j}| + r_{i_j}(m, |\vec{w}^{i_j}|) \ \text{ for } j = 1, \ldots, l & \text{by the side I.H. (3)}
\end{array}$$

Combining these estimations and using monotonicity of polynomials, one easily obtains the required estimation $|v_i^*| \leq |w_i| + (m+1) \cdot p_i(\ldots, |w_{i_j}| + r_{i_j}(m+1, |\vec{w}^{i_j}|), \ldots)$. This concludes the proof of (4) and thus the proof for the current case $\mathsf{P} \equiv \texttt{foreach}\ \mathsf{X}_j\ \texttt{[Q]}$. □

Corollary 4.3.5 (Base Bounding). *For every core program* P *with* $\mu(\mathsf{P}) = 0$ *and variables* $\vec{\mathsf{X}} := \mathsf{X}_1, \ldots, \mathsf{X}_n$ *one can find polynomials* $p_1(\vec{\mathsf{X}}), \ldots, p_n(\vec{\mathsf{X}})$ *such that for all* $\vec{w} := w_1, \ldots, w_n$*,*

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{P}\ \{|\mathsf{X}_1| \leq p_1(|\vec{w}|), \ldots, |\mathsf{X}_n| \leq p_n(|\vec{w}|)\}.$$

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 (Length bound). For stack programs P, Q such that $\mathcal{V}(\mathsf{P}) = \{\mathsf{X}_1, \ldots, \mathsf{X}_k\}$ is contained in $\mathcal{V}(\mathsf{Q}) = \mathcal{V}(\mathsf{P}) \cup \{\mathsf{Y}_1, \ldots, \mathsf{Y}_l\}$ we say that Q is a *length bound* on P, denoted $\mathsf{P} \ll \mathsf{Q}$, if

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \mathsf{P}\ \{\vec{\mathsf{X}} = \vec{v}\} \text{ and } \{\vec{\mathsf{X}} = \vec{w}, \vec{\mathsf{Y}} = \vec{u}\}\ \mathsf{Q}\ \{\vec{\mathsf{X}} = \vec{v}'\} \text{ implies } |\vec{v}| \leq |\vec{v}'|.$$

Definition 4.3.7 (Simple loops and flattened out core programs).

- A loop $\texttt{foreach}\ \mathsf{X}\ \texttt{[Q]}$ of μ-measure $n+1$ is *simple* if Q has μ-measure n.
- A core program $\mathsf{P} :\equiv \mathsf{P}_1\texttt{;}\ \ldots\ \texttt{;}\ \mathsf{P}_k$ of μ-measure $n+1$ is *flattened out* if each component P_i is either a simple loop or else $\mu(\mathsf{P}_i) \leq n$.

Given a core program P of μ-measure $n+1$, we want to construct a flattened out core program P' of μ-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* $\deg(\mathsf{P})$ for core programs P, that is, $\deg(\texttt{push(a,X)}) := 0$, $\deg(\mathsf{P}_1\texttt{;}\ \mathsf{P}_2) := \max\{\deg(\mathsf{P}_1), \deg(\mathsf{P}_2)\}$, and $\deg(\texttt{foreach X [Q]}) := 1 + \deg(\mathsf{Q})$.

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

- $\mathrm{rk}(\texttt{push(a,X)}) := 0$ for every letter $\mathsf{a} \in \Sigma$ and variable X.
- $\mathrm{rk}(\mathsf{P}_1\texttt{;}\ \mathsf{P}_2) := \max\{\mathrm{rk}(\mathsf{P}_1), \mathrm{rk}(\mathsf{P}_2)\}$.
- If P is a loop $\texttt{foreach X [Q]}$, then

$$\mathrm{rk}(\mathsf{P}) := \begin{cases} 0 & \mathsf{P} \text{ is a simple loop or } \mu(\mathsf{P}) \leq n \\ 1 + \mathrm{rk}(\mathsf{Q}) & \mathsf{Q} \text{ is a loop with } \mu(\mathsf{Q}) = n+1 \\ 1 + \sum_{i \leq k} \deg(\mathsf{Q}_i) & \mathsf{Q} \text{ is a sequence } \mathsf{Q}_0\texttt{;}\ \ldots\ \texttt{;}\ \mathsf{Q}_k \text{ without top circle} \\ & \text{and } \mu(\mathsf{Q}) = n+1. \end{cases}$$

Lemma 4.3.9 (Rank Zero). *Every core program* P *of* μ*-measure* $n+1$ *and rank* 0 *is flattened out.*

Proof. By induction on the structure of core programs P of μ-measure $n+1$ and rank 0. If P is a sequence $\mathsf{P}_1\,;\,\mathsf{P}_2$ then both components P_i have rank 0, and at least one has μ-measure $n+1$. Hence the claim follows from the induction hypothesis on the components of μ-measure $n+1$. If P is a loop of μ-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* $\mathsf{P} :\equiv \texttt{foreach}\ \mathsf{X}\ [\mathsf{Q}]$ *with* $\mu(\mathsf{P}) = n+1$ *and rank* > 0 *has a core program* P' *satisfying* $\mathsf{P} \ll \mathsf{P}'$, $\mu(\mathsf{P}') = n+1$ *and* $\mathrm{rk}(\mathsf{P}') < \mathrm{rk}(\mathsf{P})$.

Proof. Let $\mathsf{P} :\equiv \texttt{foreach}\ \mathsf{X}\ [\mathsf{Q}]$ be an arbitrary core program of μ-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 $\texttt{foreach}\ \mathsf{Y}\ [\mathsf{R}]$ with $\mu(\mathsf{P}) = \mu(\mathsf{Q}) = n+1$. In that case we define P' by

$$\mathsf{P}' :\equiv \texttt{foreach}\ \mathsf{X}\ [\texttt{foreach}\ \mathsf{Y}\ [\texttt{push(a,Z)}]];\ \texttt{foreach}\ \mathsf{Z}\ [\mathsf{R}]$$

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

Case Q is a sequence $\mathsf{Q}_0\,;\ldots;\,\mathsf{Q}_k$ without top circle, and $\mu(\mathsf{Q}_i) = \mu(\mathsf{Q}) = n+1$ for some component $\mathsf{Q}_i :\equiv \texttt{foreach}\ \mathsf{Y}\ [\mathsf{R}]$. In that case we define $\mathsf{P}' :\equiv \texttt{foreach}\ \mathsf{X}\ [\mathsf{P}_1];\ \texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_2]$ where for some *new* variable Z and any $\mathsf{a} \in \Sigma$ the core programs $\mathsf{P}_1, \mathsf{P}_2$ are defined by:

$$\mathsf{P}_1 :\equiv \mathsf{Q}_0\,;\ldots;\,\mathsf{Q}_{i-1};\ \texttt{foreach}\ \mathsf{Y}\ [\texttt{push(a,Z)}];\ \mathsf{Q}_{i+1};\ldots;\,\mathsf{Q}_k$$
$$\mathsf{P}_2 :\equiv \mathsf{Q}_0\,;\ldots;\,\mathsf{Q}_{i-1};\ \mathsf{R};\ \mathsf{Q}_{i+1};\ldots;\,\mathsf{Q}_k$$

In terms of $\mathsf{P} \ll \mathsf{P}'$, let $\#\mathsf{R}[\mathsf{P}(\vec{w})]$ denote the number of times R is executed in a run of P on $\vec{w}$. Then we conclude $\#\mathsf{R}[\mathsf{P}(\vec{w})] \leq \#\texttt{push(a,Z)}[\mathsf{P}'(\vec{w}, v)]$, for otherwise some stack $\mathsf{V} \in \mathcal{U}(\mathsf{R})$ controlled Y in $\mathsf{Q}_0\,;\ldots;\,\mathsf{Q}_{i-1};\ \mathsf{Q}_{i+1};\ldots;\,\mathsf{Q}_k$, contradicting the hypothesis that Q has no top circle. Thus $\{\vec{\mathsf{X}} = \vec{w}, \mathsf{Z} = v\}\ \texttt{foreach}\ \mathsf{X}\ [\mathsf{P}_1]\ \{|\mathsf{Z}| \geq \#\mathsf{R}[\mathsf{P}(\vec{w})]\}$, and that implies $\mathsf{P} \ll \mathsf{P}'$ by monotonicity of core programs.

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

Subcase Q_i is a simple loop, that is, $\mu(\mathsf{Q}_i) = 1 + \mu(\mathsf{R})$ and R is a sequence with top circle. First consider the *case* where Q_i is the only component of Q with μ-measure $n+1$. Then each component of P_1 is of μ-measure n, and P_2 is a sequence with a top circle. This implies $\mu(\mathsf{P}') = \mu(\texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_2]) = n+1$, and thus $\texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_2]$ is a simple loop, resulting into $\mathrm{rk}(\mathsf{P}') = \mathrm{rk}(\texttt{foreach}\ \mathsf{X}\ [\mathsf{P}_1])$. Now observe that either P_1 has a top circle, in which case $\texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_1]$ is a simple loop, or else $\mu(\mathsf{P}_1) \leq n$. In either case, we obtain $\mathrm{rk}(\mathsf{P}') = 0 < \mathrm{rk}(\mathsf{P})$. So consider the *case* where $\mu(\mathsf{Q}_j) = n+1$ for some $j \neq i$. Then both P_1 and P_2 are sequences without top circle, implying $\mu(\mathsf{P}') = \mu(\texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_2]) = n+1$ and, as R contains a loop, $\mathrm{rk}(\mathsf{P}') = \mathrm{rk}(\texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_2])$. Now $\deg(\mathsf{R}) < \deg(\mathsf{Q}_i)$ implies $\mathrm{rk}(\texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_2]) < \mathrm{rk}(\mathsf{P})$, concluding the current subcase.

Subcase Q_i is a not a simple loop, hence $\mu(\mathsf{Q}_i) = \mu(\mathsf{R})$ and R is either a loop or a sequence without top circle. In either case, P_2 has no top circle, implying $\mu(\mathsf{P}') = \mu(\texttt{foreach}\ \mathsf{Z}\ [\mathsf{P}_2]) = n+1$. Furthermore, as R contains a loop, we have $\deg(\mathsf{Q}_i) > \deg(\mathsf{R}) \geq \deg(\texttt{foreach}\ \mathsf{Y}\ [\texttt{push(a,Z)}])$ and thus $\mathrm{rk}(\mathsf{P}') < \mathrm{rk}(\mathsf{P})$, concluding the proof of the lemma. □

Lemma 4.3.11 (Flattening). *For every core program* P *of* μ*-measure* $n+1$ *one can find a flattened out core program* P′ *of* μ*-measure* $n+1$ *satisfying* $\mathtt{P} \ll \mathtt{P}'$.

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* μ*-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 $\mathtt{P}^*$ such that $\mu(\mathtt{P}) = \mu(\mathtt{P}^*)$ and $\mathtt{Q} \ll \mathtt{P}^*$ for every subprogram Q of P. Just let $\mathtt{P}^*$ result from P by recursively replacing all occurrences of imperatives `nil(X)` or `pop(X)` with `foreach X [push(b,V)]`, and all conditional statements `if top(X)≡a [Q]` with the sequence `foreach X [push(b,V)]; Q`*, for some *new* variable V and some letter $\mathtt{b} \in \Sigma$.

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* $n \to n+1$, consider any core program P of μ-measure $n+1$. We apply Flattening (4.3.11) to obtain a core program $\mathtt{P}'$ of the form $\mathtt{P}_1\,\mathtt{;}\,\ldots\,\mathtt{;}\,\mathtt{P}_k$ where each component $\mathtt{P}_i$ is either a simple loop or else $\mu(\mathtt{P}_i) \leq n$, and such that $\mathtt{P} \ll \mathtt{P}'$ and $\mu(\mathtt{P}') = n+1$. 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 $\mathtt{P}_i$ has a length bound in $\mathcal{E}^{n+3}$.

Let $\mathtt{P}_i :\equiv$ `foreach X [Q]` be any such simple loop. Hence $\mu(\mathtt{Q}) = n$ 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 $\mathtt{P}_i$. Then f_1, possibly together with other functions $f_2, \ldots, f_m$ computed by $\mathtt{P}_i$, can be defined by *simultaneous string recursion* from functions computed by Q, that is,

$$\begin{aligned} f_k(\varepsilon, \vec{w}) &= w_{k_i} \\ f_k(va, \vec{w}) &= h_k(v, \vec{w}, f_1(v, \vec{w}), \ldots, f_m(v, \vec{w})) \quad \text{for } k = 1, \ldots m. \end{aligned}$$

It follows by induction on $|v|$ that $|f_k(v, \vec{w})| \leq E^{(c\cdot|v|)}_{n+1}(\max(|v, \vec{w}|))$. 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 μ-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 μ, is presented. Thus, stack programs of μ-measure 0 compute precisely the polynomial-time computable functions.

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

Definition 4.4.1 ($\mathcal{L}^n$). For $n \geq 0$ let $\mathcal{L}^n$ denote the class of all functions f over Σ^* (for some alphabet Σ) which can be computed by a stack program of μ-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 Σ^* (for some alphabet Σ) 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}-Simulation). *For every $n \geq 0$ one can find a sequence* `LE[n+1]` *with a top circle such that $\mu($*`LE[n+1]`*$) = n$ and $\{\mathsf{Y} = w\}$* `LE[n+1]` *$\{|\mathsf{Y}| = E_{n+1}(|w|)\}$.*

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 `LE[n+2]` by:

```
LE[n+2] :≡ nil(U); foreach Y [push(a,U)];
           nil(Y); push(a,Y); push(a,Y); foreach U [LE[n+1]].
```

□

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

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

$$\{\vec{\mathsf{X}} = \vec{w}\}\ \texttt{TIME(P)}\ \{|\mathsf{V}| \leq b(|\vec{w}|)\}.$$

Hence there is a Turing machine which simulates `P` on input $\vec{w}$ in time $q_{\mathrm{time}}(b(|\vec{w}|)) \cdot b(|\vec{w}|)$, 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 `P` over $\Delta := Q \cup \Gamma \cup \{L, N, R\}$ of μ-measure n, where $\Gamma := \{\mathsf{a}_1, \ldots, \mathsf{a}_k\}$. Assume that δ consists of *moves* $\mathrm{move}_1, \ldots, \mathrm{move}_l$ where

$$\mathrm{move}_i := (\mathsf{q}_i, \mathsf{a}_i, \mathsf{q}'_i, \mathsf{a}'_i, D_i)$$

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

```
P :≡ COMPUTE-TIME-BOUND(Y);          (* of μ-measure n *)
     INITIALISE(L,Z,R);              (* of μ-measure 0 *)
     foreach Y [SIMULATE-MOVES];     (* of μ-measure 0 *)
     OUTPUT(R;O)                     (* of μ-measure 0 *)
```

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

$$(*) \qquad \{\mathsf{X} = w\}\ \texttt{INIT; SIMULATE-MOVES}^m\ \{\mathrm{L} = \alpha,\ \mathrm{reverse}(\mathsf{R}) = \mathsf{a}\beta,\ \mathsf{Z} = \mathsf{q}\}$$

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

`COMPUTE-TIME-BOUND(Y)` $:\equiv$ `nil(Y); foreach X [push(a,Y)]; LE[n+1]; ... ; LE[n+1]`

(c times) satisfies $\{\mathtt{X}{=}w\}$ `COMPUTE-TIME-BOUND(Y)` $\{\mathtt{X}{=}w, |\mathtt{Y}|{=}E^{(c)}_{n+1}(|w|)\}$.

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

$$\texttt{INITIALISE(L,Z,R)} :\equiv \texttt{nil(L); set(Z,q}_0\texttt{); REVERSE(X;R)}$$

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

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 μ-measure $\mu(\mathtt{Q})$ can be defined by

$$\begin{aligned}
&\texttt{if X}{\equiv}\varepsilon\texttt{ [Q]} :\equiv \texttt{nil(U); push(a,U); foreach X [pop(U)]; if top(U)}{\equiv}\texttt{a [Q]}\\
&\texttt{if X}{\not\equiv}\varepsilon\texttt{ [Q]} :\equiv \texttt{nil(U); push(a,U); foreach X [pop(U)]; if U}{\equiv}\varepsilon\texttt{ [Q]}
\end{aligned}$$

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

$$\begin{aligned}
\texttt{set(U,a)} \quad &\text{for} \quad \texttt{nil(U); push(a,U)}\\
\texttt{settop(U,a)} \quad &\text{for} \quad \texttt{pop(U); push(a,U)}\\
\texttt{push(top(L),R)} \quad &\text{for} \quad \texttt{if top(L)}{\equiv}\mathtt{a}_1\texttt{ [push(a}_1\texttt{,R)]; }\ldots\texttt{;}\\
&\qquad\qquad\qquad\quad \texttt{if top(L)}{\equiv}\mathtt{a}_k\texttt{ [push(a}_k\texttt{,R)]}.
\end{aligned}$$

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

$$\begin{aligned}
(R)\ &\texttt{if top(Z)}{\equiv}\mathtt{q}_i\texttt{ [if top(R)}{\equiv}\mathtt{a}_i\texttt{ [push(a}'_i\texttt{,L); set(Z,q}'_i\texttt{); pop(R);}\\
&\qquad\qquad\qquad\qquad\texttt{if R}{\equiv}\varepsilon\texttt{ [push(B,R)]]}\\
(L)\ &\texttt{if top(Z)}{\equiv}\mathtt{q}_i\texttt{ [if top(R)}{\equiv}\mathtt{a}_i\texttt{ [settop(R,a}'_i\texttt{); set(Z,q}'_i\texttt{);}\\
&\qquad\qquad\qquad\qquad\texttt{if L}{\equiv}\varepsilon\texttt{ [push(B,R)];}\\
&\qquad\qquad\qquad\qquad\texttt{if L}{\not\equiv}\varepsilon\texttt{ [push(top(L),R); pop(L)]]]}\\
(N)\ &\texttt{settop(R,a}'_i\texttt{); set(Z,q}'_i\texttt{)}
\end{aligned}$$

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

$$\begin{aligned}
\texttt{OUTPUT(R;O)} :\equiv\ &\texttt{nil(O); set(Z,a);}\\
&\texttt{foreach R [if top(R)}{\in}\Delta\setminus\Sigma\texttt{ [nil(Z)];}\\
&\qquad\texttt{if top(R)}{\in}\Sigma\texttt{ [if top(Z)}{\equiv}\texttt{a [push(top(R),O)]]]}
\end{aligned}$$

This completes the proof of the Characterisation Theorem. □

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

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

Corollary 4.4.5 (Main). *Stack programs of μ-measure n simulate exactly Turing machines which run in time bounded by a function in $\mathcal{E}^{n+2}$.*

In the proof above, given a Turing machine M with running time bounded in $\mathcal{E}^{n+2}$, we have chosen a convenient alphabet (depending on M) in order to simulate M by a stack program of μ-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 μ-measure. Thus, in particular, stack programs of μ-measure 0 compute exactly the polynomial-time computable functions.

Lemma 4.4.6 (Binary Simulation). *Every stack program* P *over* $\Sigma := \{\mathtt{a}_1, \ldots, \mathtt{a}_l\}$, $l \geq 3$, *can be* simulated *by a stack program* $\mathrm{sim}(\mathtt{P})$ *over* $\{\mathtt{0}, \mathtt{1}\}$, *preserving the μ-measure, that is,* $\mu(\mathrm{sim}(\mathtt{P})) = \mu(\mathtt{P})$, *and if* P *on input* $\vec{w}$ *outputs* v *then* $\mathrm{sim}(\mathtt{P})$ *on input* $\langle\vec{w}\rangle$ *outputs* $\langle v\rangle$, *where for* $i = 1, \ldots, L := |\mathrm{bin}(l)|$

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

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* Σ.

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

$$\begin{aligned}
\mathrm{sim}(\mathtt{nil(X)}) &:\equiv \mathtt{nil(X)} \\
\mathrm{sim}(\mathtt{pop(X)}) &:\equiv \mathtt{pop(X);}\ \ldots\ \mathtt{;pop(X)} \quad (L \text{ times}) \\
\mathrm{sim}(\mathtt{push(a}_i\mathtt{,X)}) &:\equiv \mathtt{push(}b_1^i\mathtt{,X);}\ \ldots\ \mathtt{;push(}b_L^i\mathtt{,X)}
\end{aligned}$$

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

$$\begin{array}{l}
\mathtt{nil(U);\ set(V,0);} \\
\mathtt{if\ top(X)} \equiv b_L^i\ \mathtt{[pop(X);\ push(}b_L^i\mathtt{,U);} \\
\qquad\qquad\qquad\vdots \\
\qquad\quad \mathtt{if\ top(X)} \equiv b_1^i\ \mathtt{[pop(X);\ push(}b_1^i\mathtt{,X);}\ \ldots\ \mathtt{;push(}b_L^i\mathtt{,X);}\ \mathrm{sim}(\mathtt{Q})\mathtt{;\ set(V,1)]} \ldots \mathtt{]} \\
\mathtt{if\ top(V)} \equiv \mathtt{0\ [foreach\ U\ [if\ top(U)} \equiv \mathtt{0\ [push(0,X)];} \\
\qquad\qquad\qquad\qquad\qquad \mathtt{if\ top(U)} \equiv \mathtt{1\ [push(1,X)]]]}
\end{array}$$

Observe that stack V is set to 1 if and only if the top L letters in X match the binary code $\langle \mathtt{a}_i\rangle$. 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 $\mathtt{P}_1\mathtt{;\ P}_2$. Here we define $\mathrm{sim}(\mathtt{P})$ as the sequence $\mathrm{sim}(\mathtt{P}_1)\mathtt{;}\ \mathrm{sim}(\mathtt{P}_2)$.

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

$$\begin{array}{l}
\mathrm{sim}(\mathtt{P}) :\equiv \mathtt{nil(Y);\ nil(V);} \\
\qquad\quad \mathtt{foreach\ X\ [if\ |V|} < L\ \mathtt{[push(1,V);\ push(top(X),Y)];} \\
\qquad\qquad\qquad \mathtt{if\ |V|} \geq L\ \mathtt{[}\mathrm{sim}(\mathtt{Q})^*\mathtt{;\ nil(Y);\ nil(V)]]}
\end{array}$$

where $\mathrm{sim}(\mathtt{Q})^*$ results from $\mathrm{sim}(\mathtt{Q})$ by replacing recursively each occurrence of a conditional statement $\mathrm{sim}(\mathtt{if\ top(X)} \equiv \mathtt{a}_j\ \mathtt{[R]})$ with conditional $\mathtt{if\ TOP(Y)} \equiv_L b_L^j \ldots b_1^j\ \mathtt{[}\mathrm{sim}(R)^*\mathtt{]}$ (see above), and

where the conditionals `if |V|` $< L$ `[...]` and `if |V|` $\geq L$ `[...]` are implemented as follows:

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

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 `nil(X)`, `suc(X)` and `pred(X)` by sequence and loop statements, where `X` is a variable. We assume an infinite supply of variables $\texttt{X}, \texttt{Y}, \texttt{Z}, \texttt{O}, \texttt{U}, \texttt{V}$, 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 μ on loop programs and relate it to the Grzegorczyk hierarchy by showing that loop programs of μ-measure n compute exactly the functions in $\mathcal{E}^{n+2}$.

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

- Every *imperative* among `nil(X)`, `suc(X)`, `pred(X)` is a loop program.
- If $\texttt{P}_1, \texttt{P}_2$ are loop programs, then so is the *sequence* statement $\texttt{P}_1\texttt{; P}_2$.
- If `P` is a loop program with no occurrence of `suc(X)`, `nil(X)` or `pred(X)`, then so is the *loop* statement `loop X [P]`.

Again we use $\mathcal{V}(\texttt{P})$ to denote the set of variables occurring in `P`.

Note 4.5.2. *Any loop program can be written uniquely in the form* $\texttt{P}_1\texttt{; } \ldots \texttt{; P}_k$ *such that each* $\texttt{P}_i$ *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 `nil(X)` sets register `X` to zero. The imperative `suc(X)` increments the number stored in `X` by one, while `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 `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 `loop X [P]`.

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

$$\mathcal{U}(\texttt{P}) := \{\texttt{X} \mid \texttt{P contains an imperative suc(X)}\}$$
$$\mathcal{C}(\texttt{P}) := \{\texttt{X} \mid \texttt{P contains a loop loop X [Q] with } \mathcal{U}(\texttt{Q}) \neq \emptyset\}$$

Now the concept of "control" for stack programs (cf. 4.2.1) passes on to loop programs. As well, the *measure* μ *on loop programs* is defined as on stack programs, that is, $\mu(\texttt{imp}) := 0$ for every imperative `imp`, $\mu(\texttt{Q}_1\texttt{; Q}_2) := \max\{\mu(\texttt{Q}_1), \mu(\texttt{Q}_2)\}$, and

$$\mu(\texttt{loop X [Q]}) := \begin{cases} \mu(\texttt{Q}) + 1 & \text{if Q is a sequence with a top circle} \\ \mu(\texttt{Q}) & \text{else} \end{cases}$$

where $\mathtt{Q} \equiv \mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_l$ has a *top circle* if there exists a component $\mathtt{Q}_i$ with $\mu(\mathtt{Q}_i) = \mu(\mathtt{Q})$ such that some $\mathtt{Y}$ controls some $\mathtt{Z}$ in $\mathtt{Q}_i$, and $\mathtt{Z}$ controls $\mathtt{Y}$ in $\mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_{i-1}\mathtt{;}\ \mathtt{Q}_{i+1}\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_l$.

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

$$\mathtt{PRED(X)} :\equiv \mathtt{nil(Y);nil(Z);loop\,X\,[nil(Y);loop\,Z\,[suc(Y)];suc(Z)]}$$

Thus, the loop program `loop U [PRED(X);nil(X);loop Y [suc(X)]]` computes modified subtraction. But observe that this program has μ-measure 1, and it appears that without the primitive instruction `pred(X)` there were no loop program of μ-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 μ-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* $\mathtt{P}$ *with registers* $\vec{\mathtt{X}} := \mathtt{X}_1, \ldots, \mathtt{X}_l$ *has a* stack simulation $\mathcal{I}(\mathtt{P})$ *with stacks* $\vec{\mathtt{X}}$ (*over the unary alphabet* $\{\mathtt{a}\}$) *such that*

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

(b) $\{\vec{\mathtt{X}} = \vec{x}\}\,\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{y}\}$ *if and only if* $\{\mathtt{X}_1 = \mathtt{a}^{x_1}, \ldots, \mathtt{X}_l = \mathtt{a}^{x_l}\}\,\mathcal{I}(\mathtt{P})\,\{\mathtt{X}_1 = \mathtt{a}^{y_1}, \ldots, \mathtt{X}_l = a^{y_l}\}$.

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

$$\begin{aligned}
\mathcal{I}(\mathtt{nil(X)}) &:\equiv \mathtt{nil(X)}\\
\mathcal{I}(\mathtt{suc(X)}) &:\equiv \mathtt{push(a,X)}\\
\mathcal{I}(\mathtt{pred(X)}) &:\equiv \mathtt{pop(X)}\\
\mathcal{I}(\mathtt{P}_1\mathtt{;}\ \mathtt{P}_2) &:\equiv \mathcal{I}(\mathtt{P}_1)\mathtt{;}\ \mathcal{I}(\mathtt{P}_2)\\
\mathcal{I}(\mathtt{loop\,X\,[Q]}) &:\equiv \mathtt{foreach\,X\,[}\mathcal{I}(\mathtt{Q})\mathtt{]}
\end{aligned}$$

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

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

Proof. Let $\mathtt{P}$ be any loop program of μ-measure n, and let f be any k-ary function computed by $\mathtt{P}$. Consider the corresponding function $\mathcal{I}(f)$ computed by the stack simulation $\mathcal{I}(\mathtt{P})$ obtained from Lemma 4.5.3. By part (a) of that lemma $\mathcal{I}(\mathtt{P})$ has μ-measure n, and part (b) implies $f(x_1, \ldots, x_k) = |\mathcal{I}(f)(\mathtt{a}^{x_1}, \ldots, \mathtt{a}^{x_k})|$ 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 $|\mathcal{I}(f)(\vec{w})| \leq b(|\vec{w}|)$ for all $\vec{w}$. Now, as $|\mathtt{a}^y| = y$, we conclude that b is a bound on f. □

Theorem 4.5.4 implies that every function computed by a loop program of μ-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 μ-measure 0 in the sense that `P[f]` on $\vec{x}, v$ outputs $f(\vec{x})$ whenever $v \geq E^{(m)}_{n+1}(\max(\vec{x}))$, for some constant m. As the function $E^m_{n+1} \circ \max$ is computable by a loop program `ME[n+1]` of μ-measure n, the sequence `ME[n+1]; P[f]` has μ-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 $\mathtt{P} :\equiv \mathtt{P}_1\mathtt{;}\ldots\mathtt{;}\mathtt{P}_k$ and variables $\mathtt{X}, \mathtt{Y}$ we implement the *conditional statement* `if X≤Y then [P]` as the sequence $\mathtt{Q}_1\mathtt{;}\ \mathtt{Q}_2\mathtt{;}\ \mathtt{Q}_3$ defined from *new variables* $\mathtt{U}^*, \mathtt{V}^*$ as follows:

$$\begin{aligned}
\mathtt{Q}_1 &:\equiv \mathtt{nil(U^*);\ loop\ X\ [suc(U^*)];\ loop\ Y\ [pred(U^*)]}\\
\mathtt{Q}_2 &:\equiv \mathtt{nil(V^*);\ suc(V^*);\ loop\ U^*\ [pred(V^*)]}\\
\mathtt{Q}_3 &:\equiv \mathtt{loop\ V^*\ [P_1];}\ \ldots\ \mathtt{;\ loop\ V^*\ [P_k]}
\end{aligned}$$

Lemma 4.5.6 (Conditional). *Let* `P` *be a loop program, and let* $\mathtt{Z}_0, \ldots, \mathtt{Z}_n$ *be the variables* $\mathcal{V}(\mathtt{P}) \cup \{\mathtt{X}, \mathtt{Y}\}$ *such that* $\mathtt{Z}_i = \mathtt{X}$ *and* $\mathtt{Z}_j = \mathtt{Y}$. *Then* $\mathtt{Q} :\equiv$ `if X≤Y then [P]` *satisfies:*

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

(b) $\mu(\mathtt{Q}) = \mu(\mathtt{P})$.

(c) *For any* $\mathtt{X}_0, \mathtt{X}_1 \in \{\mathtt{Z}_0, \ldots, \mathtt{Z}_n\}$, $\mathtt{X}_0$ *controls* $\mathtt{X}_1$ *in* `Q` *if and only if* $\mathtt{X}_0$ *controls* $\mathtt{X}_1$ *in* `P`.

Proof. Observe that $\mathtt{Q}_3$ could not be simply `loop V* [P]`, because that could result in $\mu(\mathtt{Q}) > \mu(\mathtt{P})$. Hence (b), (c) are obvious. For (a), assume $\{\vec{\mathtt{Z}} = \vec{z}\}\ \mathtt{P}\ \{\vec{\mathtt{Z}} = \vec{u}\}$. As $\{\vec{\mathtt{Z}} = \vec{z}\}\ \mathtt{Q}_1\ \{\vec{\mathtt{Z}} = \vec{z}, \mathtt{U}^* = z_i \dot{-} z_j\}$, we obtain $\{\vec{\mathtt{Z}} = \vec{z}\}\ \mathtt{Q}_1\mathtt{;}\ \mathtt{Q}_2\ \{\vec{\mathtt{Z}} = \vec{z}, \mathtt{V}^* = 1 \dot{-} (z_i \dot{-} z_j)\}$ and hence (a), since $z_i \leq z_j \Leftrightarrow \mathtt{V}^* = 1$, and $z_i > z_j \Leftrightarrow \mathtt{V}^* = 0$. □

The naïve approach to compute $f \in \mathcal{E}^{n+2}$ by a loop program of μ-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

$$\mathtt{Y = X} \ :\equiv\ \mathtt{nil(Y);\ loop\ X\ [suc(Y)]}$$

can cause new top circles and result into a μ-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)

$$\mathtt{Y\ +=\ X} \ :\equiv\ \mathtt{loop\ X\ [suc(Y)]}$$

(read *increase* `Y` *by* `X`) by the *simulation* (each using *new* variables $\mathtt{U}, \mathtt{W}, \mathtt{H}$)

$$\begin{aligned}
\mathtt{SIM(Y\ +=\ X)} &:\equiv \mathtt{nil(U);\ loop\ V\ [suc(U);\ if\ U{\leq}X\ [suc(Y)]]}\\
\text{with } \mathtt{if\ U{\leq}X\ [suc(Y)]} &:\equiv \mathtt{W = U;\ loop\ X\ [pred(W)];}\\
&\qquad\quad \mathtt{nil(H);\ suc(H);\ loop\ W\ [pred(H)];}\\
&\qquad\quad \mathtt{loop\ H\ [suc(Y)]}
\end{aligned}$$

Observe that if the number stored in `V` is greater or equal to that stored in `X`, then `SIM(Y += X)` increases `Y` by `X` when executed – as `Y += X` does. Moreover, `X` does not control `Y` in `SIM(Y += X)`, unlike `Y += X`. Of course, now `V` controls `Y`, and all other variables $\mathtt{U}, \mathtt{W}, \mathtt{H}$ *local* to `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 `SIM(Y = X)` below, and facts about it listed in $(*)$.

$$\mathtt{SIM(Y = X)} :\equiv \mathtt{nil(Y);\ SIM(Y\ +=\ X)}$$

($*$) $\mathtt{SIM(Y=X)}$ is a circle-free sequence such that $\mu(\mathtt{SIM(Y=X)})=0$, as the control in $\mathtt{SIM(Y=X)}$ is $\{\mathtt{U}\to\mathtt{W}, \mathtt{H}\to\mathtt{Y}, \mathtt{V}\to\mathtt{U}, \mathtt{V}\to\mathtt{W}, \mathtt{V}\to\mathtt{H}, \mathtt{V}\to\mathtt{Y}\}$, for some *local* variables $\mathtt{U},\mathtt{H},\mathtt{W}$. In particular, $\mathtt{V}\notin\mathcal{U}(\mathtt{SIM(Y=X)})$ and $\mathtt{X},\mathtt{Y}\notin\mathcal{C}(\mathtt{SIM(Y=X)})$. Furthermore, if the number stored in $\mathtt{V}$ is greater or equal to that stored in $\mathtt{X}$, then $\mathtt{SIM(Y+=X)}$ increases $\mathtt{Y}$ by $\mathtt{X}$ when executed.

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

(a) $f(\vec{x})\le E_{n+1}^{(m_f)}(\max(\vec{x}))$ *for all $\vec{x}$,*

(b) $\{\vec{\mathtt{X}}=\vec{x},\mathtt{V}=v\}\,\mathtt{P[f]}\,\{\vec{\mathtt{X}}=\vec{x},\mathtt{V}=v,\mathtt{O}=f(\vec{x})\}$ *for all $\vec{x},v$ with $v\ge E_{n+1}^{(m_f)}(\max(\vec{x}))$,*

(c) $\mathtt{P[f]}$ *is a sequence without a (top) circle, $\mathtt{V}\notin\mathcal{U}(\mathtt{P[f]})$, $\vec{\mathtt{X}},\mathtt{O}\notin\mathcal{C}(\mathtt{P[f]})$, and $\mu(\mathtt{P[f]})=0$.*

Proof. Induction on the structure of $f\in\mathcal{E}^{n+2}$ where $n\ge 0$.

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 $\mathtt{P[f]}:\equiv\mathtt{nil(O)}$ and $m_f:=0$.

– If f is S then define $\mathtt{P[f]}:\equiv\mathtt{SIM(O=X_i);\ suc(O)}$ and $m_f:=0$.

– If f is Π_i^m then define $\mathtt{P[f]}:\equiv\mathtt{SIM(O=X_i)}$ and $m_f:=0$.

– If f is $\max$ then define $\mathtt{P[f]}:\equiv\mathtt{SIM(0=X_1);\ if\ X_1\le X_2\ [SIM(0=X_2)]}$ 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

$$\mathtt{P[f]}:\equiv\mathtt{SIM(L=X);\ nil(O);\ loop\,L\,[SIM(O+=X)];\ suc(O);\ suc(O)}$$

for some *new* variable $\mathtt{L}$. 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 $f(\vec{x})=h(g_1(\vec{x}),\dots,g_k(\vec{x}))$ with $h,g_1,\dots,g_k\in\mathcal{E}^{n+2}$. The induction hypothesis yields simulations $\mathtt{P[h]},\mathtt{P}[g_1],\dots,\mathtt{P}[g_k]$ with witnesses $m_h,m_1,\dots,m_k$, respectively. Thus, assuming distinct variables $\mathtt{O},\mathtt{O}_1,\dots,\mathtt{O}_k$, we have $\{\vec{\mathtt{X}},\mathtt{V}=\vec{x},v\}\,\mathtt{P}[g_i]\,\{\vec{\mathtt{X}},\mathtt{V}=\vec{x},v,\mathtt{O}_i=g_i(\vec{x})\}$ whenever $v\ge E_{n+1}^{(m_i)}(\max(\vec{x}))$, and $\{\vec{\mathtt{O}},\mathtt{V}=\vec{y},v\}\,\mathtt{P[h]}\,\{\vec{\mathtt{O}},\mathtt{V}=\vec{y},v,\mathtt{O}=h(\vec{y})\}$ for $v\ge E_{n+1}^{(m_h)}(\max(\vec{y}))$. 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 $\mathtt{V},\vec{\mathtt{X}}$, the following will do:

$$\mathtt{P[f]}:\equiv\mathtt{P}[g_1];\dots;\mathtt{P}[g_k];\mathtt{P[h]}$$
$$m_f:=m_h+\max(m_1,\dots,m_k)$$

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

$$\mathtt{P}:\equiv\ \mathtt{P[g];\ L=Y;\ nil(Y);\ loop\,L\,[Z=O;\ P[h];\ suc(Y)]}$$

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 $v \geq E_{n+1}^{(m_g)}(\max(\vec{x}))$ and $v \geq E_{n+1}^{(m_h)}(\max(\vec{x}, i, f(\vec{x}, i)))$ for $i < y$, the latter following from

$$\begin{aligned} v &\geq E_{n+1}^{(m_f)}(\max(\vec{x}, y)) \\ &\geq E_{n+1}^{(m_h)}(\max(\vec{x}, i, E_{n+1}^{(m_b)}(\max(\vec{x}, i)))) \\ &\geq E_{n+1}^{(m_h)}(\max(\vec{x}, i, f(\vec{x}, i))). \end{aligned}$$

Thus, we conclude $\{\vec{\mathsf{X}}, \mathsf{Y}, \mathsf{V} = \vec{x}, y, v\}\, \mathtt{P}\, \{\vec{\mathsf{X}}, \mathsf{Y}, \mathsf{V} = \vec{x}, y, v, \mathsf{O} = f(\vec{x})\}$.

Now observe that P contains the top circle $\mathsf{Y} \to \mathsf{L} \to \mathsf{Y}$. 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 μ-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 $\mathsf{V}, \vec{\mathsf{X}}, \mathsf{O}$, we obtain `P[f]` by modifying P to

$$\texttt{P[f]} :\equiv \texttt{P[g]; SIM(L = Y); nil(Y); loop L [SIM(Z = O); P[h]; suc(Y)]}.$$

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 $\{\mathsf{X} = x\}$* `E[n+1]` *$\{\mathsf{X} = E_{n+1}(x)\}$ 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

$$\texttt{E[n+2]} :\equiv \texttt{Y = X; nil(X); suc(X); suc(X); loop Y [E[n+1]]}.$$

□

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

Proof. First we treat the implication "⇒". So let P be any loop program of μ-measure n. By Loop Bounding (4.5.4) there is a function in $\mathcal{E}^{n+2}$ 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 $\mathcal{E}^{n+2}$. All cases are obvious, except possibly the case where P is of the form `loop X [Q]`. In that case we make use of the fact that $\mathcal{E}^{n+2}$ is closed under bounded simultaneous recursion (cf. Background 2.2).

As for "⇐", consider any $f \in \mathcal{E}^{n+2}$. Loop Simulation (4.5.7) yields a program `P[f]` and a witness m_f such that $\{\vec{\mathsf{X}} = \vec{x}, \mathsf{V} = v\}$ `P[f]` $\{\mathsf{O} = f(\vec{x})\}$ for $v \geq E_{n+1}^{(m_f)}(\max(\vec{x}))$. Let `E[n+1]` be the program of μ-measure n satisfying $\{\mathsf{V} = y\}$ `E[n+1]` $\{\mathsf{V} = E_{n+1}(y)\}$ obtained from E_{n+1}-Computation (4.5.8). Using conditional statements one defines a program M of μ-measure 0 satisfying $\{\vec{\mathsf{X}} = \vec{x}\}\, \mathtt{M}\, \{\vec{\mathsf{X}} = \vec{x}, \mathsf{V} = \max(\vec{x})\}$. Thus, the sequence `M; E[n+1]; ...; E[n+1]; P[f]` (m_f times) has μ-measure n and computes f. □

Theorem 4.5.9 ensures that the measure μ is sound with respect to computational complexity of *functions* computed by loop programs. This implies that μ is also sound with respect to *computational complexity* of loop programs, that is, for every loop program P of μ-measure n the function $\text{TIME}_{\mathtt{P}}$ belongs to $\mathcal{E}^{n+2}$, where $\text{TIME}_{\mathtt{P}}(\vec{x})$ is the number of imperatives executed in a run of P on $\vec{x}$. To see this, given any loop

program P of μ-measure n, let $\mathtt{P}^{\#}$ result from P by replacing every occurrence of an imperative `imp` with `imp; suc(V)`, where V is any fixed *new* variable. Then `TIME(P)`$:\equiv$`nil(V);` $\mathtt{P}^{\#}$ has μ-measure n and computes $\text{TIME}_{\mathtt{P}}$, that is, $\{\vec{\mathtt{X}}=\vec{x}\}\,\mathtt{TIME(P)}\,\{\mathtt{V}=\text{TIME}_{\mathtt{P}}(\vec{x})\}$. Hence $\text{TIME}_{\mathtt{P}} \in \mathcal{E}^{n+2}$ 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 μ-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 FP. 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 FP, 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; if <test>[Q]` where R is a program which runs in polynomial time, `<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 $\nu\colon L\text{-programs} \to \mathbb{N}$.

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

- *sound* if every L-program of ν-measure 0 is feasible,
- *complete* if every honestly feasible L-program has ν-measure 0, and
- *adequate* if every function in FP is L-computable with ν-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 μ be the measure on core programs as defined in section 4. The next theorem is good news.

Theorem 4.6.4. *The pair (μ, 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 $\mu(\mathsf{P})>0$. Then P contained a loop `foreach X [Q]` where Q is a sequence $\mathsf{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathsf{Q}_l$ 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 $\mathsf{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathsf{Q}_{i-1}\mathtt{;}\ \mathsf{Q}_{i+1}\mathtt{;}\ \ldots\ \mathtt{;}\ \mathsf{Q}_l$. 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 (μ, C) is obviously not adequate. As core programs are length-monotonic, there are plenty of functions in FP which are not C-computable, let alone C-computable with μ-measure 0. However, wouldn't it be nice if we could extend (μ, C) to an adequate pair and still preserve both soundness and completeness? Well, it is not possible.

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

Proof. Let $\{\rho_i\}_{i\in\omega}$ 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 $f_n \in$ 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 ρ_i halts. Since (ν, L) is adequate and sound, there is an honestly feasible program $\mathsf{Q} \in L$ of ν-measure 0 such that

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

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 :≡ foreach X [Q; foreach Z [foreach V [push(1,W)]];
                   foreach W [push(1,V)]]
```

is also in L, where X, V, W are *new* stacks. Now, if ρ_n never halts then `foreach V [push(1,W)]` will never be executed, whatever the inputs to P. Thus, if ρ_n never halts, then P is honestly feasible. In contrast, if ρ_n halts after s steps, say, then part `foreach V [push(1,W)]` and part `foreach W [push(1,V)]` will be executed each time the body of the outermost loop is executed whenever $\mathsf{Y} = y$ with $|y| \geq s$. Each such execution implies that the height of stack V is at least doubled. Thus, if ρ_n eventually halts, then P is not feasible. In other words, P is honestly feasible if and only if ρ_n never halts. As P is effectively constructible from n, we conclude that (ν, L) cannot be complete. For if (ν, L) were complete, then ρ_n would never halt if and only if $\nu(\mathsf{P})=0$. This would yield an algorithm which decides whether ρ_n halts: Construct P from n and then check whether $\nu(\mathsf{P})>0$. 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 $\vec{\mathbf{n}}$, one can compute the normal form $\mathsf{nf}(t\vec{\mathbf{n}})$ in time $p_t(|\vec{\mathbf{n}}|)$. 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 $\mathsf{nf}(t\vec{\mathbf{n}})$: One is to normalise $t\vec{x}$ to u, say, which may take a long time (super-polynomial in the length of t), the other is to normalise $u[\vec{\mathbf{n}}/\vec{x}]$, which will take polynomial time in the length of $\vec{\mathbf{n}}$. 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 (V below) of an outer recursion to that (v below) of an inner recursion. For example, the following "binary version" $\mathcal{B}$ of Ackermann's variant $\mathcal{A}$ can be defined by:

$$t_{\mathcal{B}} := \mathbf{RN}_{\iota\to\iota}\,\mathbf{s}_1\,(\lambda u^{\iota} V^{\iota\to\iota} y^{\iota}.\,\mathbf{RN}_{\iota}\,y\,(\lambda u^{\iota} v^{\iota}.Vv)\,(\mathbf{s}_1 y))$$

One easily verifies $|\mathcal{B}(m,n)| = \mathcal{A}(|m|,|n|)$, hence $\mathcal{B}$ is non-primitive recursive, too. Another problem is that by nesting computed functionals in recursions exponential growth rate is obtained by a single higher-type recursion on notation. For example, the "binary version" d of the exponentially growing function e, satisfying $|d(m,n)| = e(|m|,|n|) = 2^{|m|} + |n|$, can be defined by:

$$t_d := \mathbf{RN}_{\iota\to\iota}\,\mathbf{s}_1\,(\lambda u^{\iota} V^{\iota\to\iota} y^{\iota}.V(Vy))$$

Thus, to set up a subsystem RA of Gödel'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,

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

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 σ is only admitted in the form $\mathbf{RN}_{\sigma}\,g\,h\,n$ for !-free types σ, called *safe* types, and for h, n of complete types $!(!\iota \to \sigma \to \sigma)$, $!\iota$ 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 σ is admitted only if x^σ is ***affinable*** in r, that is, either r has at most one free occurrence of x^σ, or else r has a subterm a of type ι with a single free occurrence of x^σ such that every free occurrence of x^σ 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^σ 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 ι, and inductively, if σ, τ are types, then so are $\sigma \to \tau$ and $!\sigma$.
- The *level* $l(\sigma)$ of σ 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 τ.

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.

$\mathbf{0}$	ι		
$\mathbf{s}_0$	$\iota \to \iota$		(binary successor $x \mapsto 2{\cdot}x$)
$\mathbf{s}_1$	$\iota \to \iota$		(binary successor $x \mapsto 2{\cdot}x + 1$)
$\mathbf{p}$	$\iota \to \iota$		(binary predecessor $x \mapsto \lfloor \frac{x}{2} \rfloor$)
$\mathbf{c}_\sigma$	$\iota \to \sigma \to \sigma \to \sigma \to \sigma$	for σ safe	(cases in type σ)

Terms are built from these constants and typed variables x^σ by introduction and elimination rules for the two type forms $\sigma \to \tau$ and $!\sigma$, and by formation of recursion terms[1], that is,

$$(\lambda x^\sigma.r^\tau)^{\sigma\to\tau}, \quad (r^{\sigma\to\tau} s^\sigma)^\tau, \quad (!r^\sigma)^{!\sigma}, \quad (\kappa r^{!\sigma})^\sigma, \quad (\mathbf{RN}_\rho\, g^\rho\, h^{!(!\iota\to\rho\to\rho)}\, n^{!\iota})^\rho \text{ for safe } \rho.$$

[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 $\mathbf{RN}_\sigma$ as a term former rather than as a constant.

Recall that a *binary numeral* is either $\mathbf{0}$ or it is of the form $\mathbf{s}_{i_1} \dots \mathbf{s}_{i_k} \mathbf{s}_1 \mathbf{0}$ with $i_j \in \{0,1\}$ for $j = 1, \dots, k$, $k \geq 0$. Binary numerals distinct from $\mathbf{0}$ are denoted by $\mathbf{s}_i\mathbf{n}$. The *conversion rules* are then as expected:

$$\begin{array}{rl} (\lambda x.r)s \mapsto r[s/x] & \mathbf{c}_\sigma \mathbf{0}\, r\, t_0\, t_1 \mapsto r \\ \kappa(!r) \mapsto r & \mathbf{c}_\sigma(\mathbf{s}_i\mathbf{n})\, r\, t_0\, t_1 \mapsto t_i \\ \mathbf{s}_0\mathbf{0} \mapsto \mathbf{0} & \mathbf{RN}_\sigma g\, h\, !\mathbf{0} \mapsto g \\ \mathbf{p0} \mapsto \mathbf{0} & \mathbf{RN}_\sigma g\, h\, !(\mathbf{s}_i\mathbf{n}) \mapsto \kappa h\, !(\mathbf{s}_i\mathbf{n})\, (\mathbf{RN}\, g\, h\, !\mathbf{n}) \\ \mathbf{p}(\mathbf{s}_i\mathbf{n}) \mapsto \mathbf{n} & \end{array}$$

The *length* $|t|$ of a term t is defined by $|x| = |c| = 1$; $|\lambda x.r| = |!r| = |r\kappa| = |r| + 1$; $|rs| = |r| + |s| + 1$.

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 $t \longrightarrow t'$ if t' results from t by converting one redex; we say t *reduces to* t', or t' *is a reduct of* t, if $t \longrightarrow^* t'$ where $\longrightarrow^*$ denotes as usual the reflexive and transitive closure of $\longrightarrow$. Observe that $\longrightarrow$ 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 $\mathsf{nf}(t)$ in normal form.

One writes $\mathsf{FV}(t)$ for the set of free variables of t, and $\mathsf{FO}(x,t)$ 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 σ, since we are only interested in the computational behaviour of terms. Accordingly, $\mathrm{H}_{!\sigma} := \mathrm{H}_\sigma$, and the value $[\![t]\!]_\varphi$ of a term t in an environment φ 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 λy binds a variable free in either a_i or a_j, then both a_i and a_j lie within the scope of the λy.

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^ι with $\mathsf{FO}(x,a) = 1$ 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 $\mathsf{FO}(x,r) \leq 1$.

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:

($!\,\mathrm{I}$) If r^σ is an RA term with $\mathsf{FV}(r)$ all complete, then $(!r)^{!\sigma}$ is an RA term.

(A) If $r^{\sigma\to\tau}$ and s^σ are RA terms, then $(rs)^\tau$ is an RA term; provided that if r is complete, then $\mathsf{FV}(s)$ are all complete.

(L) If r^τ is an RA term, then $(\lambda x^\sigma.r)^{\sigma\to\tau}$ is an RA term; provided that if σ is incomplete, then x is affinable in a reduct of r.

Observe that every variable of type ι is trivially affinable in every term, because it is an affination of itself. Thus the restriction on lambda abstracted variables x^σ 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:

$$\begin{aligned} t_\oplus &:= \lambda x^\iota y^{!\iota}.\mathbf{RN}_\iota\, x\, !(\lambda u^{!\iota} v^\iota.\, \mathbf{c}\,(\kappa u)\,\mathbf{0}\,(\mathbf{s}_0 v)\,(\mathbf{s}_1 v))\, y \\ t_\otimes &:= \lambda x^{!\iota} y^{!\iota}.\mathbf{RN}_\iota\, \mathbf{0}\, !(\lambda u^{!\iota} v^\iota.\, t_\oplus\, v\, x)\, y \end{aligned}$$

In system RA computed functionals in higher-type recursions can only be passed to safe affinable input positions. Admitting recursion $\mathbf{RN}_\sigma$ for incomplete σ would result in exponential growth. For example, writing the equations $d'(0,n)=n$ and $d'(s_i(m),n)=d'(m,d'(m,\mathrm{sq}(n)))$ for $s_i(m)\neq 0$, with $\mathrm{sq}(n)=2^{2|n|}$, as a recursion in type $!\iota\to\iota$, one could define the function d' satisfying $d'(m,n)\geq n^{2^{|m|}}$. To see this, define sq by $t_{\mathrm{sq}}:=\lambda y^{!\iota}.\mathbf{RN}_\iota\,(\mathbf{s}_1\mathbf{0})\,!(\lambda u^{!\iota} v^\iota.\,\mathbf{s}_0(\mathbf{s}_0 v))\,y$, and then d' by

$$t_{d'} := \lambda x^{!\iota}.\mathbf{RN}_{!\iota\to\iota}\,(\lambda y^{!\iota}.\kappa y)\,!(\lambda u^{!\iota} V^{!\iota\to\iota} y^{!\iota}.\, V\,!(t_{\mathrm{sq}}\, y))\, x.$$

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 $\mathsf{FO}(x,r)\leq 1$ 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 $r\longrightarrow r'$, then x need not be affinable in r'. For example, suppose that x has an affination a in r such that $\mathsf{FO}(x,r)\geq 2$, and r' is obtained from r by reducing a single occurrence of a to some a' such that $\mathsf{FO}(x,a')=1$ and a' is no subterm of a, (for example, $a=\mathbf{c}_\sigma(\mathbf{s}_i\mathbf{n})\,r\,t_0\,t_1\,\vec{r}$ with $x\in\mathsf{FV}(t_i)$ and $a'=t_i\,\vec{r}$), say

$$r=\ldots a\ldots a\ldots a\ldots \longrightarrow \ldots a\ldots a'\ldots a\ldots = r'.$$

Then neither $\mathsf{FO}(x,r')\leq 1$ 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 $r''=\ldots a'\ldots a'\ldots a'\ldots$ 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 $r\longrightarrow r'$, 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:

$$\lambda x^{!\iota}.\mathbf{RN}_{\iota\to\iota}\,\mathbf{s}_1\,(\lambda u^{!} V^{\iota\to\iota} y^\iota.(\lambda y^\iota.Vy)(Vy))\, x$$

Without scope equivalence, the two occurrences of Vy 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 $\mathsf{FV}(r)$ all are complete by $(!\,\mathrm{I})$. If r is κs, then s has type $!\sigma$, hence $\mathsf{FV}(r)$ all are complete by the induction hypothesis on s. If r is $s^{\sigma\to\tau}t^{\sigma}$, 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 $\mathsf{FV}(r)$ all are complete by the induction hypothesis on s. □

Lemma 5.3.2 (Closure under Substitution). *If $r, \vec{s}$ are* RA *terms, then so is $r[\vec{s}/\vec{x}]$.*

Proof. We proceed by induction on the structure of r. If r is a variable or a constant, then $r[\vec{s}/\vec{x}]$ is either some s_i, or else it is r. In either case, $r[\vec{s}/\vec{x}]$ is an RA term.

Case r is $!s$. Then $r[\vec{s}/\vec{x}] = !s[\vec{s}/\vec{x}]$ and by $(!\,\mathrm{I})$ $\mathsf{FV}(s)$ all are complete. Now $s[\vec{s}/\vec{x}]$ 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 $(!\,\mathrm{I})$ $!s[\vec{s}/\vec{x}]$ is in RA.

Case r is κs. Then $r[\vec{s}/\vec{x}] = \kappa s[\vec{s}/\vec{x}]$, so the claim follows from the induction hypothesis on s.

Case r is $s^{\sigma\to\tau}t^{\sigma}$. Then $r[\vec{s}/\vec{x}] = (s[\vec{s}/\vec{x}])(t[\vec{s}/\vec{x}])$, and both $s[\vec{s}/\vec{x}]$ and $t[\vec{s}/\vec{x}]$ are RA terms by the induction hypothesis. We may assume that s is complete, hence by (A) $\mathsf{FV}(t)$ 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 $t[\vec{s}/\vec{x}]$ has complete free variables only. Now (A) gives that $r[\vec{s}/\vec{x}]$ is in RA.

Case r is $\lambda x.s$. We may assume that $x \notin \vec{x}$, and $x \notin \mathsf{FV}(s_i)$ for all s_i. Hence $r[\vec{s}/\vec{x}] = \lambda x.s[\vec{s}/\vec{x}]$, and $s[\vec{s}/\vec{x}]$ 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 x is affinable in the reduct $r''[\vec{s}/\vec{x}]$ of $r'[\vec{s}/\vec{x}]$, showing that $r[\vec{s}/\vec{x}]$ is in RA.

Case t is a recursion $\mathbf{RN}_{\sigma}ghn$. We conclude $t \in \mathrm{RA}$ from the induction hypothesis on g, h, n. □

Lemma 5.3.3 (Affinability). *Let r be an* RA *term such that x is affinable in r. If $r \longrightarrow r'$, then x is affinable in a reduct of r'.*

Proof. Induction on the structure of r, assuming $r \longrightarrow r'$ and that x is affinable in r, and $\mathsf{FO}(x, r) \geq 1$.

Case $\mathsf{FO}(x, r) = 1$. We may assume $\mathsf{FO}(x, r') \geq 2$. By Completeness (5.3.1) the only conversion rule capable of multiplying an incomplete variable is a β 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 $\mathsf{FO}(x, r) \geq 2$. 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 $\mathsf{FO}(x, r) \geq 2$, 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 $\mathsf{FO}(x, r') \geq 2$. Inspecting all possible forms of R other than a β redex, we see by Completeness (5.3.1) that in either case, a is still an affination of x in

r'. 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 $y \notin \mathsf{FV}(a)$. Thus, if a occurs in t, we conclude from $\mathsf{FO}(x, r') \geq 2$ that a is still an affination of x in r' – where the occurrences of a in r are multiplied in r' in case $\mathsf{FO}(y, s) \geq 2$. It remains to consider the case where a occurs in s. If $y \in \mathsf{FV}(a)$, 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 $y \notin \mathsf{FV}(a)$, then a is still an affination of x in r'.

□

Theorem 5.3.4 (Closure under Reduction). *If $r \longrightarrow r'$ 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 $r \longrightarrow r'$ 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 $\mathsf{FV}(s)$ all are complete, and r' is $!s'$ such that $s \longrightarrow s'$. The side induction hypothesis on s yields that s' is in RA. Hence r' is in RA, as $\mathsf{FV}(s') \subseteq \mathsf{FV}(s)$.

Case r is κs. Then either s is $!s'$ and r' is s', in which case we are done, or else r' is $\kappa s'$ and $s \longrightarrow s'$. In the latter case, the side induction hypothesis on s implies that r' is an RA term.

Case r is $s^{\sigma\to\tau}t^{\sigma}$. 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 $\mathbf{s}_0$, $\mathbf{p}$, κ or $\mathbf{c}$ conversions, and for β conversions we apply Substitution (5.3.2). In case of an $\mathbf{RN}$ conversion, it suffices to consider the rule $\mathbf{RN}_\sigma g\, h\, !(\mathbf{s}_i\mathbf{n}) \mapsto \kappa h\, !(\mathbf{s}_i\mathbf{n})\, (\mathbf{RN}_\sigma g\, h\, !\mathbf{n})$. Observe that κh 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 $s \longrightarrow s'$. 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:

$$\begin{array}{l} s \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_n \equiv s^*,\ n \geq 1, \text{ and } x \text{ is affinable in } s^* \\ \downarrow \\ s' \end{array}$$

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 $\mathsf{nf}(s^*) = \mathsf{nf}(s) = \mathsf{nf}(s')$ 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 $\mathsf{nf}(t)$. For the proof, we proceed by induction on the height of the reduction tree for t. Suppose that $t \longrightarrow t'$ 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 $\mathsf{nf}(t'') = \mathsf{nf}(t)$.

Case r is $\mathbf{RN}_\sigma ghn$. Then either r' results from r by converting that outermost $\mathbf{RN}$ redex, or else r' results from r by reducing a component of r. The former case has been treated above, in the latter case $r' \in \mathrm{RA}$ follows from the induction hypothesis on the reduced component. □

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 $\mathsf{nf}(r)$. In particular, if r is closed and ground, then $\mathsf{nf}(r)$ 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 κr is a normal term, then r is of the form $\kappa(\kappa\cdots\kappa(X\vec{r}_1)\vec{r}_2\ldots)\vec{r}_n$ for some complete variable X.*

Proof. Induction on the structure of normal terms r. For type reasons, r is of the form $U\vec{s}$ where U is neither a constant nor a symbol $\mathbf{RN}_\sigma$. Since κr is normal, U is not the symbol !. If U is a variable X, then it has a type $\vec{\sigma}\to!\tau$ and we are done. If U is κ, 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 $\kappa\vec{s}=r$. □

Definition 5.4.2. For types σ,ρ we say that *σ occurs strictly left in ρ*, denoted $\sigma\prec\rho$, if either $\rho=\rho_1\to\rho_2$ and $(\sigma\preceq\rho_1$ or $\sigma\prec\rho_2)$, or else $\rho=!\rho'$ and $\sigma\prec\rho'$. As usual $\preceq$ stands for $\prec$ or $=$.

Lemma 5.4.3 (Type Analysis). *Let t^σ be a normal term, and let x^τ be a higher-type non-safe variable of t. Then $(*)$ $\tau\prec\sigma$ or $\tau\preceq\rho$ for some $y^\rho\in FV(t)$.*

Proof. We proceed by induction on the structure of normal terms t of any type σ.

Case t is $\lambda z^{\sigma_1}.r^{\sigma_2}$. If x is z, then $\tau=\sigma_1\prec(\sigma_1\to\sigma_2)=\sigma$. Otherwise the induction hypothesis on r yields either $\tau\prec\sigma_2$, implying $\tau\prec(\sigma_1\to\sigma_2)=\sigma$, or else $\tau\preceq\rho$ for some $y^\rho\in\mathsf{FV}(r)$. In the latter case, if $y\neq z$ then $y\in\mathsf{FV}(t)$, and if y is z, then $\tau\preceq\sigma_1\prec\sigma$. In either case, $(*)$ is true of t.

Case t is $X\vec{r}$. If $\#\vec{r}=0$ or X is x, then the second alternative of $(*)$ holds. Otherwise x occurs in some $r_i^{\sigma_i}$, and so the induction hypothesis yields $\tau\prec\sigma_i$, implying $\tau\prec\mathrm{type}(X)$, or else $\tau\preceq\rho$ for some $y^\rho\in\mathsf{FV}(r_i)\subseteq\mathsf{FV}(t)$. In either case, the second alternative of $(*)$ holds.

Case t is $c\vec{r}$ where c is either a constant or $\mathbf{RN}_\sigma$. Then either x occurs in some safe or ground type subterm, or else c is $\mathbf{RN}_\sigma$ and x occurs in the step term r_2 of type $!(!\iota\to\rho)$ for the safe type $\rho:=\sigma\to\sigma$. In either case, we conclude from the induction hypothesis and from the hypothesis on τ being higher-type and non-safe that the second alternative of $(*)$ is true of t. This is obvious in all cases, except possibly where c is $\mathbf{RN}_\sigma$. Here observe $\tau\prec!(!\iota\to\rho)\Leftrightarrow\tau\prec(!\iota\to\rho)\Leftrightarrow\tau\preceq!\iota$ or $\tau\preceq\rho$.

Case t is $!r^\sigma$. 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 $y^\rho\in\mathsf{FV}(r)=\mathsf{FV}(t)$. In either case, $(*)$ is true of t.

Case t is $\kappa\vec{s}$. 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 $(*)$ is true of t. For either x is X, or else x occurs in some r_{ij} of type σ_{ij}, in which case the induction hypothesis gives either $\tau\prec\sigma_{ij}$, implying $\tau\prec\mathrm{type}(X)$, or else $\tau\preceq\rho$ for some $y^\rho\in\mathsf{FV}(r_{ij})\subseteq\mathsf{FV}(t)$. □

Corollary 5.4.4 (Structure). *Let t be a normal* RA *term of ground type or of type $\sigma=\vec{\sigma}\to\tau$ where $\vec{\sigma},\tau$ all are safe or ground, and assume that $\mathsf{FV}(t)$ all are safe or ground.*

(a) *t contains safe or ground type variables only.*

(b) *Every subterm κr of t is of the form $\kappa\vec{\kappa}y$ 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 $\mathbf{RN}_\sigma$ 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 κr of t, then κr is normal (as t is normal), and we may apply Kappa (5.4.1) to obtain that r is of the form $\kappa(\ldots\kappa(X\vec{r}_1)\vec{r}_2\ldots)\vec{r}_n$ for some complete variable X. Now part (a) implies that X is a complete ground type variable y. Hence $\#\vec{r}_1 = \ldots = \#\vec{r}_n = 0$ and κr is $\kappa\vec{\kappa}y$.

As for (c), given a subterm rs of t, then rs is normal (as t is normal), hence rs is either of the form $U\vec{r}$ with U being a constant, or else it is of the form $\mathbf{RN}_\sigma g\, h\, n\, \vec{s}$. 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. □

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 $\vec{\sigma} \to \tau$ with $\vec{\sigma}, \tau$ all ground, one can find a polynomial p_t such that for all numerals $\vec{\mathbf{n}}$ of types $\vec{\sigma}$ one can compute the numeral $\mathsf{nf}(t\vec{\mathbf{n}})$ in time $p_t(\sum|\mathbf{n}_i|)$. *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 $\vec{\mathbf{n}}$ 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 κs 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 β 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 $\mathsf{FO}(x,s) \leq 1$, 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 affination a of x in s, $u = \ldots a \ldots a \ldots$ say, by $\lambda x.\ldots(\lambda y^\iota.\ldots y \ldots y \ldots)a \ldots$.

Definition 5.5.2. SRA-*terms* are simple RA terms r such that

(S) every subterm $\lambda x.s$ with x higher-type (safe) satisfies $\mathsf{FO}(x,s) \leq 1$.

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 $\beta(t) \longrightarrow^* t$.*

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 $\mathsf{FO}(x,r) \geq 2$. By assumption x has an affination 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 $z \in \mathsf{FV}(a)$, 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'. □

Note. Every SRA-term t can be written uniquely in *head form*, being of the form $U\vec{r}$ where U is a variable (safe or ground), a constant or $!s$, κs (ground) or $\mathbf{RN}_\sigma g\, h\, n$, or else U is of the form $\lambda x.s$ with $\mathsf{FO}(x, s) \leq 1$, or else with $\mathsf{FO}(x, s) > 1$ and x ground. Call $\vec{r}$, 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 $!\iota \to \sigma$, then t is of the form $\lambda x^{!\iota}.s$.*

(c) *If t has type $!(!\iota \to \sigma)$, then t is of the form $!\,\lambda x^{!\iota}.s$.*

Proof. Obviously, (c) follows from (a), (b). As for (a), let $U\vec{r}$ 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 $\mathbf{RN}_\sigma g\, h\, n$, for all these have safe types. Furthermore, as t is higher-type, U cannot be κs, because by simplicity (5.5.1) κs is ground, and so were t. Finally, as t has type $!\sigma$, U cannot be $\lambda x.s$, because that would imply $\#\vec{r} \geq 1$, hence by simplicity (5.5.1) Ur_1 were safe, and so were $U\vec{r}$. Thus, the only possible form for U is $!s$, implying $\#\vec{r}{=}0$ and thus $t = !s$ as required.

As for (b), let $U\vec{r}$ be the head form of t with type $!\iota \to \sigma$. By simplicity U is neither a variable nor of the form κs. Furthermore, for type reasons U is neither a constant nor of the form $\mathbf{RN}_\sigma g\, h\, n$ or $!r$. Thus, U must be $\lambda x.s$, and in this case $\#\vec{r}{=}0$ follows. For otherwise simplicity (5.5.1) would imply that Ur_1 were safe, and so were $U\vec{r}$, contradicting that t has type $!\iota \to \sigma$. □

The system SRA is not closed under $\mathbf{RN}$ conversion and β conversions $(\lambda y.s)t$ with $\mathsf{FO}(y, s) \geq 2$ 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 β 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 $t\vec{v}$, κt, $!t$, $(\lambda x.t)\vec{v}$, or $(t[s/x])\vec{v}$, where t, x and s are any components of the given terms and $\vec{v}$ 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_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 contain a constant or a new variable; test on the head form and branch; test on the head form and perform 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 $\mathsf{FO}(x,s) > 1$ and formation of r and $s\vec{r}$;

– test on $(\lambda x.s)r\vec{r}$ with $\mathsf{FO}(x,s) \leq 1$ and form the term $(s[r/x])\vec{r}$;

– test on $\mathbf{c}_\sigma t_1 t_2 t_3 t_4 \vec{r}$, 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 $\vec{\mathbf{n}};\vec{x}$ such that each $\mathbf{n}_i$ is a numeral.

Theorem 5.5.6 (Base Evaluation). *For every* $\mathbf{RN}$*-free* SRA*-term* t *of ground type and numeral environment* $\vec{\mathbf{n}};\vec{x}$ *such that* $\mathsf{FV}(t) \subseteq \vec{x}$*,*

(1) *one can compute* $\mathsf{nf}(t[\vec{\mathbf{n}}/\vec{x}])$ *in at most* $2|t|$ *steps,*

(2) *the number of used registers is* $\leq |t| + \#\vec{\mathbf{n}}$*, and*

(3) *every term* s *occurring in the computation satisfies* $|s| \leq |t| + \max|\mathbf{n}_i|$.

Proof. We describe the algorithm nf, which at input $t,\vec{\mathbf{n}};\vec{x}$ outputs $\mathsf{nf}(t,\vec{\mathbf{n}};\vec{x}) = \mathsf{nf}(t[\vec{\mathbf{n}}/\vec{x}])$ 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_i or a constant, or U is $!s$ or κs, or else U is $\lambda x.s$ with either $\mathsf{FO}(x,s)\leq 1$, or else $\mathsf{FO}(x,s)>1$ and x is ground.

If t is x_i, then output $\mathbf{n}_i$. We have performed two steps, and (2), (3) are obvious.

If t is $\mathbf{0}$, then output $\mathbf{0}$. We have performed two steps, and (2), (3) are obvious.

If t is Ur where U is one of the symbols $\mathbf{s}_1, !$, first compute $\mathbf{n} := \mathsf{nf}(r,\vec{\mathbf{n}};\vec{x})$, then form $U\mathbf{n}$. We have performed $\leq 2 + 2|r| \leq 2|t|$ steps, using $\leq 1 + |r| + \#\vec{\mathbf{n}} \leq |t| + \#\vec{\mathbf{n}}$ registers, and (3) follows.

If t is $\mathbf{s}_0 r$, first compute $\mathbf{n} := \mathsf{nf}(r,\vec{\mathbf{n}};\vec{x})$, then output $\mathbf{0}$ if $\mathbf{n}$ is $\mathbf{0}$, otherwise form $\mathbf{s}_0\mathbf{n}$. We have performed at most $3 + 2|r| \leq 2|t|$ steps, using at most $1 + |r| + \#\vec{\mathbf{n}} \leq |t| + \#\vec{\mathbf{n}}$ registers. Concerning (3), we obtain $|\mathbf{s}_0\mathbf{n}| \leq 2 + |r| + \max|\mathbf{n}_i| = |t| + \max|\mathbf{n}_i|$.

If t is $\mathbf{p}r$, first compute $\mathbf{n} := \mathsf{nf}(r,\vec{\mathbf{n}};\vec{x})$, then form $\mathbf{m}$ if $\mathbf{n}$ is $\mathbf{s}_i\mathbf{m}$, else output $\mathbf{0}$. Properties (1), (2), (3) follow as in the previous case.

If t is κr, first compute $!\mathbf{n} := \mathsf{nf}(r,\vec{\mathbf{n}};\vec{x})$, then output $\mathbf{n}$. We have performed $\leq 1 + 2|r| \leq 2|t|$ steps, using $\leq |r| + \#\vec{\mathbf{n}}$ registers, and (3) follows directly from the induction hypothesis on r.

If t is $\mathbf{c}_\sigma s t_0 t_1 t_2 \vec{r}$, first compute $\mathbf{n} := \mathsf{nf}(s,\vec{\mathbf{n}};\vec{x})$, then compute $\mathsf{nf}(t_j\vec{r},\vec{\mathbf{n}};\vec{x})$ where j is 0 if $\mathbf{n}$ is $\mathbf{0}$, and where j is $i+1$ if $\mathbf{n}$ is of the form $\mathbf{s}_i\mathbf{m}$. We have performed $\leq 2 + 2|s| + 2|t_j\vec{r}| \leq 2|t|$ steps, using at most $1 + \max(|s| + \#\vec{\mathbf{n}}, |t_j\vec{r}| + \#\vec{\mathbf{n}}) \leq |t| + \#\vec{\mathbf{n}}$ registers. (3) follows from the induction hypothesis on $t_j\vec{r}$.

If t is $(\lambda x.s)r\vec{r}$ with $\mathsf{FO}(x,s)>1$, first compute $\mathbf{n} := \mathsf{nf}(r,\vec{\mathbf{n}};\vec{x})$, then copy $\mathbf{n}$ to u_x, and finally compute $\mathsf{nf}(s\vec{r},\vec{\mathbf{n}},\mathbf{n};\vec{x},x)$. Observe that $\vec{\mathbf{n}},\mathbf{n};\vec{x},x$ is a numeral environment such that $\mathsf{FV}(s\vec{r}) \subseteq \vec{x},x$. 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 $\mathsf{FO}(x,s)\leq 1$, compute $\mathsf{nf}((s[r/x])\vec{r},\vec{\mathbf{n}};\vec{x})$. As $|(s[r/x])\vec{r}| < |t|$, we have performed at most $1 + 2|(s[r/x])\vec{r}| \leq 2|t|$ steps, using $\leq |(s[r/x])\vec{r}| + \#\vec{\mathbf{n}}$ registers, and (3) is obvious. □

Corollary 5.5.7 (Base Normalisation). *For every closed* $\mathbf{RN}$*-free* SRA*-term* t *of ground type one can compute* $\mathsf{nf}(t)$ *in at most* $2|t|$ *steps, using* $\leq |t|$ *registers, and every term occurring in the computation has a length* $\leq |t|$*.* □

In order to compute $\mathbf{RN}$-free SRA-terms $\mathsf{rf}(t, \vec{n}; \vec{x})$ in a given environment $\vec{n}; \vec{x}$, we slightly generalise the technique above. To "unfold" every recursion $\mathbf{RN}ghn$ in t, we require that $\mathsf{FV}(t[\vec{n}/\vec{x}])$ 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 $\mathsf{nf}(\mathsf{rf}(n[\vec{n}/\vec{x}]))$. Of course, this only works for $\mathbf{RN}$-free $\vec{n}$. To make the induction work, however, we require that t is safe or ground. Finally, to ensure that $\mathsf{rf}(t, \vec{n}; \vec{x})$ is in SRA, we show in addition that no safe variable $y \in \mathsf{FV}(t) \setminus \mathsf{FV}(\vec{n})$ is multiplied in $\mathsf{rf}(t, \vec{n}; \vec{x})$. 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* $\vec{n}$ $\mathbf{RN}$*-free and* $\mathsf{FV}(t[\vec{n}/\vec{x}])$ *all safe, one can compute an* $\mathbf{RN}$*-free* SRA*-term* $\mathsf{rf}(t, \vec{n}; \vec{x})$ *satisfying* $t[\vec{n}/\vec{x}] \longrightarrow^* \mathsf{rf}(t, \vec{n}; \vec{x})$ *such that*

- (i) *the number of steps, the number of used registers and the length of every term occurring in the computation all are* $\leq q_t(\sum |n_i|)$*, and*
- (ii) *for every safe variable* $y \in \mathsf{FV}(t) \setminus \mathsf{FV}(\vec{n})$ *one has* $\mathsf{FO}(y, \mathsf{rf}(t, \vec{n}; \vec{x})) \leq \mathsf{FO}(y, t)$*.*

Proof. By induction of $|t|$. Let $\vec{n}; \vec{x}$ be a fixed but arbitrary environment with $\vec{n}$ all $\mathbf{RN}$-free and $\mathsf{FV}(t[\vec{n}/\vec{x}])$ all safe. Let m be $\sum |n_i|$. We write #steps, #registers and maxlength for the three quantities above, and call their maximum *bound*.

If t is some x_i, then output n_i. We define $q_t := 2 + \mathsf{id}$, and (i), (ii) are obvious.

If t is $\lambda x.r$, first compute $R := \mathsf{rf}(r, \vec{n}; \vec{x})$, then form $T := \lambda x.R$. Observe that both x and r are safe, for t has safe type. Therefore, as $\mathsf{FV}(t[\vec{n}/\vec{x}])$ all are safe, so are $\mathsf{FV}(r[\vec{n}/\vec{x}])$, 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 $y \in \mathsf{FV}(r) \setminus \mathsf{FV}(\vec{n})$ 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 $x \in \mathsf{FV}(r) \setminus \mathsf{FV}(\vec{n})$, hence $\mathsf{FO}(x, \mathsf{rf}(r, \vec{n}; \vec{x})) \leq \mathsf{FO}(x, r)$. Thus, as t, R are in SRA, so is T.

If t is $Ur_1 \dots r_l$ where U is a variable distinct from all $\vec{x}$, or U is one of the symbols $\mathbf{0}, \mathbf{s}_i, \mathbf{p}, \mathbf{c}_\sigma, \kappa$, first compute each $R_i := \mathsf{rf}(r_i, \vec{n}; \vec{x})$, then form $T := UR_1 \dots R_l$. Observe that by Simplicity (5.5.1) each r_i is safe or ground, and that $\mathsf{FV}(r_i[\vec{n}/\vec{x}])$ 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 $R_1, \dots, R_l$.

If t is $(\lambda x.s)rr_1 \dots r_l$, 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 $\mathsf{FV}(t[\vec{n}/\vec{x}])$ all are safe, so are $\mathsf{FV}(r[\vec{n}/\vec{x}])$ and each $\mathsf{FV}(r_i[\vec{n}/\vec{x}])$, and $\mathsf{FV}(s[\vec{n}/\vec{x}])$ all are safe provided that x is safe.

Subcase x is safe. First compute $S := \mathsf{rf}(s, \vec{n}; \vec{x})$, $R := \mathsf{rf}(r, \vec{n}; \vec{x})$ and each $R_i := \mathsf{rf}(r_i, \vec{n}; \vec{x})$, then form $T := (\lambda x.S)RR_1 \dots R_l$. 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 $S, R, R_1, \dots, R_l$.

Subcase x is complete. First compute $n := \mathsf{rf}(r, \vec{n}; \vec{x})$, copy n to u_x, then compute $\mathsf{rf}(sr_1 \dots r_l, \vec{n}, n; \vec{x}, x)$. Observe that $t' := sr_1 \dots r_l$ is a safe SRA-term with $|t'| < |t|$ and $\mathsf{FV}(t'[\vec{n}/\vec{x}])$ all safe. Hence we can apply the induction hypothesis to define $q_t := |t| + q_r + q_{t'} \circ (\mathsf{id} + q_r)$. 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)

$\mathsf{FV}(r[\vec{n}/\vec{x}])$ all are complete, and so are $\mathsf{FV}(n)$, since $r[\vec{n}/\vec{x}] \longrightarrow^* n$. Therefore, if $y \in \mathsf{FV}(t)\setminus\mathsf{FV}(\vec{n})$ is safe, then $y \in \mathsf{FV}(t')\setminus\mathsf{FV}(\vec{n}, n)$, and thus (ii) for t follows from (ii) for t'.

Remaining case t is of the form $\mathbf{RN}ghnr_1 \dots r_l$. By Quasi-Normality (5.5.4) h is of the form $!(\lambda x^{!\iota}.H)$ for some safe H. In this case we will output the SRA-term

$$\mathsf{rf}(t, \vec{n}; \vec{x}) := (T_0(T_1 \dots (T_{k-1}G) \dots))R_1 \dots R_l$$

where $G, T_0, \dots, T_{k-1}, R_1, \dots, R_l$ are defined as follows:

$$\begin{aligned} G &:= \mathsf{rf}(g, \vec{n}; \vec{x}) \\ k &:= |[\![\mathbf{n}]\!]| \text{ with } !\mathbf{n} := \mathsf{nf}(\mathsf{rf}(n, \vec{n}; \vec{x})) \\ T_i &:= \mathsf{rf}(H, \vec{n}, !\mathbf{n}_i; \vec{x}, x) \text{ with } \mathbf{n}_i \text{ obtained from } \mathbf{n} \text{ by deleting the first } i \text{ leading constants } \mathbf{s}_0, \mathbf{s}_1 \\ R_j &:= \mathsf{rf}(r_j, \vec{n}; \vec{x}) \end{aligned}$$

First we compute the numeral $!\mathbf{n}$. Since n has type $!\iota$, 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 $\mathbf{RN}$-free SRA-term $\mathsf{rf}(n, \vec{n}; \vec{x})$ with bound $\leq q_n(m)$, for $n[\vec{n}/\vec{x}] \longrightarrow^* \mathsf{rf}(n, \vec{n}; \vec{x})$. Then by Base Normalisation (5.5.7) one obtains the numeral $!\mathbf{n} := \mathsf{nf}(\mathsf{rf}(n, \vec{n}; \vec{x})) = \mathsf{nf}(n[\vec{n}/\vec{x}])$ with

$$\#\text{steps} \leq 2|\mathsf{rf}(n, \vec{n}; \vec{x})| \leq 2q_n(m), \quad \#\text{registers} \leq q_n(m), \quad \text{maxlength} \leq q_n(m).$$

We now compute the term $T_0(T_1 \dots (T_{k-1}G) \dots)$ by an obvious loop with $k \leq |\mathbf{n}| \leq q_n(m)$ 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 $G := \mathsf{rf}(g, \vec{n}; \vec{x})$ with bound $\leq q_g(m)$ in a result register u, and consider the register w holding $\mathbf{n} = \mathbf{n}_0$ as counter. If w holds $\mathbf{n}_i = \mathbf{0}$, output the content of register u. Otherwise, w holds $\mathbf{n}_i \neq \mathbf{0}$ and u holds $(T_{k-i} \dots (T_{k-1}G) \dots)$. In that case compute $!\mathbf{n}_{k-i-1}$ from $\mathbf{n}$ and $\mathbf{n}_i$ in the environment register u_x; this clearly is possible with some bound $q_1(|\mathbf{n}|) \leq q_1(q_n(m))$ for some polynomial q_1. Compute $T_{k-i-1} := \mathsf{rf}(H, \vec{n}, !\mathbf{n}_{k-i-1}; \vec{x}, x)$ in v, with bound $q_H(m + |\mathbf{n}_{k-i-1}|) \leq q_H(m + q_n(m))$. Update u by applying the content of v onto u's original content. This gives $T_{k-i-1}(T_{k-i} \dots (T_{k-1}G) \dots)$ in one step, with no additional register and maxlength increased by $|T_{k-i-1}| \leq q_H(m + q_n(m))$. Finally, update w to hold $\mathbf{n}_{i+1}$ and go to the initial test of the loop.

Let us now estimate our bound. We go $k \leq |\mathbf{n}| \leq q_n(m)$ times through the loop. From above we obtain:

$$\#\text{steps in each round} \leq 1 + q_1(|\mathbf{n}|) + q_H(m + |\mathbf{n}_{k-i-1}|) + 2 \leq 1 + q_1(q_n(m)) + q_H(m + q_n(m)) + 2$$

The number of registers used is 3 (for v, u, u_x) plus $q_1(q_n(m))$ (to compute $!\mathbf{n}_{k-i-1}$) plus $q_H(m + q_n(m))$ (to compute T_{k-i-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

$$\big(3 + q_1(q_n(m)) + q_H(m + q_n(m))\big) \cdot q_n(m) + q_g(m).$$

Finally, in an obvious loop with l rounds, compute $R_j := \mathsf{rf}(r_j, \vec{n}; \vec{x})$ with bound $q_{r_j}(m)$, assuming u holds the term $(T_0(T_1 \dots (T_{k-1}G) \dots))R_1 \dots R_{j-1}$, 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

$$q_t(m) := |t| + 2q_n(m) + \big(3 + q_1(q_n(m)) + q_H(m + q_n(m))\big) \cdot q_n(m) + q_g(m) + \sum_{i=1}^{l} q_{r_i}(m).$$

It remains to verify (ii) for $\mathsf{rf}(t, \vec{n}; \vec{x})$. By Completeness (5.3.1) no T_i contains a safe free variable, because h has complete type and $H[\vec{n}, !\mathbf{n}_i/\vec{x}, x] \longrightarrow^* T_i$ 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), $\mathbf{RN}$-Elimination (5.5.8) and Base Normalisation (5.5.7), for each closed RA term t of type $\vec{\sigma} \to \tau$ with $\vec{\sigma}, \tau$ all ground, we obtain a polynomial-time evaluation strategy which at input $\vec{\mathbf{n}}$ outputs the numeral $\mathsf{nf}(t\vec{\mathbf{n}})$.

First we clarify on the role of safe input positions σ_i for the case where τ 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 τ is complete to that where τ is safe (ι).

Lemma 5.5.9 (Redundancy). *Let t be any normal RA term of type $\sigma_1, \ldots, \sigma_k \to !\tau$ such that $\mathsf{FV}(t)$ and $\sigma_1, \ldots, \sigma_k, \tau$ all are ground. Let φ, φ' be any environments differing only on safe free variables of t, and let $\vec{\mathbf{n}}, \vec{\mathbf{m}}$ be any numerals of types $\sigma_1, \ldots, \sigma_k$ which differ only on safe components σ_i. Then $[\![t\vec{\mathbf{n}}]\!]_\varphi = [\![t\vec{\mathbf{m}}]\!]_{\varphi'}$.*

Proof. We proceed by induction on t. By Structure (5.4.4) all variables of t are safe or ground, and all subterms κs are ground. Therefore, as t has complete type, t cannot be of the form $U\vec{r}$ with U a safe variable or a constant or a symbol $\mathbf{RN}$. If t is κr or $!r$, 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 $[\![t]\!]_\varphi = [\![t]\!]_{\varphi'}$ follows from the hypothesis on φ, φ'. 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 φ, φ' and $\vec{\mathbf{n}}, \vec{\mathbf{m}}$, straightforward semantical reasoning verifies $[\![t\vec{\mathbf{n}}]\!]_\varphi = [\![t\vec{\mathbf{m}}]\!]_{\varphi'}$. □

Theorem 5.5.10 (Bounding). *Let t be a closed RA term of type $\sigma_1, \ldots, \sigma_k \to \tau$, where $\sigma_1, \ldots, \sigma_k, \tau$ all are ground. Then one can find a polynomial p_t such that for all numerals $\mathbf{n}_1, \ldots, \mathbf{n}_k$ with types $\sigma_1, \ldots, \sigma_k$ respectively, one can compute the numeral $\mathsf{nf}(t\vec{\mathbf{n}})$ in time $p_t(\sum_i |\mathbf{n}_i|)$.*

Proof. One must find a polynomial p_t such that for all numerals $\vec{\mathbf{n}}$ of types $\vec{\sigma}$, one can compute $\mathsf{nf}(t\vec{\mathbf{n}})$ in time $p_t(m)$ where $m := \sum_i |\mathbf{n}_i|$. Let $\vec{x}$ be new variables of types $\vec{\sigma}$. We consider two cases.

Case τ 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 $\vec{\mathbf{n}}$. 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 $t' := \beta(\mathsf{nf}(t\vec{x}))$ satisfying $t' \longrightarrow^* \mathsf{nf}(t\vec{x})$. Let c be the number of "steps" needed to compute t'. By $\mathbf{RN}$-Elimination (5.5.8) one obtains a closed $\mathbf{RN}$-free SRA-term $\mathsf{rf}(t', \vec{n}; \vec{x})$ such that $t'[\vec{\mathbf{n}}/\vec{x}] \longrightarrow^* \mathsf{rf}(t', \vec{\mathbf{n}}; \vec{x})$. This requires at most $q_{t'}(m)$ steps, and uses at most this many registers of this size. As the output is in a register, this also bounds the length $|\mathsf{rf}(t', \vec{\mathbf{n}}; \vec{x})|$. Using Base Normalisation (5.5.7) one obtains the numeral $\mathsf{nf}(\mathsf{rf}(t', \vec{\mathbf{n}}; \vec{x})) = \mathsf{nf}(t\vec{\mathbf{n}})$ in a total of $c + 3q_{t'}(m)$ 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, q_{time} say, we have thus computed the numeral $\mathsf{nf}(t\vec{\mathbf{n}})$ in time $p_t(m) := c' + 3q_{t'}(m) \cdot q_{\mathsf{time}}(q_{t'}(m))$ for some constant c'.

Case τ is complete. By Redundancy (5.5.9) all safe input positions σ_i of t are redundant. This implies $[\![\mathsf{nf}(t\vec{\mathbf{n}})]\!] = [\![\mathsf{nf}(t\vec{\mathbf{n}}')]\!]$ where $\vec{\mathbf{n}}'$ results from $\vec{\mathbf{n}}$ by replacing each safe $\mathbf{n}_i$ with $\mathbf{0}$. Thus, it suffices to compute $\mathsf{nf}(t\vec{\mathbf{n}}')$. So let $t\vec{x}'$ be the RA term where $\vec{x}'$ results from $\vec{x}$ by replacing each safe x_i with $\mathbf{0}$, and let $\vec{\mathbf{n}}'', \vec{x}''$ result from $\vec{\mathbf{n}}, \vec{x}$ respectively by cancelling all safe components. Then $[\![t\vec{\mathbf{n}}']\!] = [\![t\vec{x}'[\vec{\mathbf{n}}''/\vec{x}'']]\!] = [\![t'[\vec{\mathbf{n}}''/\vec{x}'']]\!]$ where t' is $\beta(\mathsf{nf}(t\vec{x}'))$. Therefore, the argument for the first case carries over to $t\vec{x}'$ with respect to $\vec{\mathbf{n}}''; \vec{x}''$. In fact, one obtains the same bound p_t. □

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 FPTIME-functions into the system RA. One could use any of the resource-free function algebra characterisations [5, 6, 46] of FPTIME; we pick [5].

Theorem 5.6.1 (Embedding FPTIME). *Every polynomial-time computable function is denoted by an* RA *term.*

Proof. Recall (cf. Background 2.8) the characterisation of 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 $!\vec{\iota}; \vec{\iota} \to \iota$ 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 $\mathbf{0}, \mathbf{s}_i, \mathbf{p}$ respectively. If f is the binary conditional $c(y_1,y_2,y_3)$, then let t_f be $\lambda y_1y_2y_3.\mathbf{c}_\iota y_1y_2y_2y_3$. If f is a projection $\pi^{m,n}_j$, then let t_f be $\lambda x_1 \ldots x_{m+n}.u_j$ where u_j is κx_j if $j \leq m$, otherwise u_j is x_j.

Case $f(\vec{x};\vec{y}) = g(\vec{g}(\vec{x};);\vec{h}(\vec{x};\vec{y}))$ 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 ι.

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

$$\begin{aligned} h(0,\vec{x};\vec{y},z) &= 0 \\ h(s_i(x),\vec{x};\vec{y},z) &= h_i(x,\vec{x};\vec{y},z) \quad \text{for } s_i(x) \neq 0. \end{aligned}$$

Using the induction hypothesis to obtain t_{h_0} and t_{h_1}, we can define h by cases:

$$t_h := \lambda x\vec{x}\vec{y}z.\, \mathbf{c}_\iota\, (\kappa x)\, \mathbf{0}\, (t_{h_0}(\mathbf{p}\kappa x)\vec{x}\vec{y}z)\, (t_{h_1}(\mathbf{p}\kappa x)\vec{x}\vec{y}z))$$

where $x, \vec{x}$ all are of type $!\iota$, and $\vec{y}, z$ all are of type ι. Concluding the current case, we define t_f by

$$\lambda x\vec{x}.\, \mathbf{RN}_{\vec{\iota}\to\iota}\, (t_g\vec{x})\, !(\lambda u^{!\iota} V^{\vec{\iota}\to\iota}\vec{y}.\, t_h u\vec{x}\vec{y}(V\vec{y}))\, x.$$

In each case, one easily verifies $[\![t_f]\!] = f$. □

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 μ, which assigns in a purely syntactic fashion to each program P in the language under consideration a natural number $\mu(\mathsf{P})$.

We proved that the measure μ 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 μ 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 μ-measure 0 compute exactly the polynomial-time computable functions, and loop programs of μ-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 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 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 μ 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 $\mathsf{X} := \mathsf{Y}$ 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

[1] K. Aehlig, J. Johannsen, H. Schwichtenberg, and S. A. Terwijn. Linear ramified higher type recursion and parallel computation. In *PTCS*, volume 2183. LNCS, Springer-Verlag, 2001.

[2] J. L. Bates and R. L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[3] S. J. Bellantoni. *Predicative Recursion and Computational Complexity*. PhD thesis, Toronto, September 1993.

[4] S. J. Bellantoni. Predicative recursion and the polytime hierarchy. In P. Clote and J. Remmel, editors, *Feasible Mathematics II, Perspectives in Computer Science*, pages 15–29. Birkhäuser, 1994.

[5] S. J. Bellantoni and S. Cook. A New Recursion-Theoretic Characterization of the Polytime Functions. *Computational Complexity*, 2:97–110, 1992.

[6] S. J. Bellantoni and K.-H. Niggl. Ranking Primitive Recursions: The Low Grzegorczyk Classes Revisited. *SIAM Journal of Computing*, 29(2), 1999.

[7] S. J. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.

[8] U. Berger. Program extraction from normalization proofs. In M. Bezem and J.F. Groote, editors, *Typed Lambda Calculi and Applications*, Springer Lecture Notes in Computer Science Logic, Volume 664, pages 91–106. Springer, 1993.

[9] U. Berger and H. Schwichtenberg. Program development by proof transformation. In H. Schwichtenberg, editor, *Proof and Computation*, Series F: Computer and Systems Sciences, pages 1–46, Berlin, 1995. NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20 – August 1, 1993, Springer.

[10] U. Berger, H. Schwichtenberg, and M. Seisenberger. From Proofs to Programs in the Minlog System. The Warshall Algorithm and Higman's Lemma. Submitted, 1997.

[11] S. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4(2):175–205, 1994.

[12] S. Buss. *Bounded Arithmetic*. PhD thesis, Princeton University, 1985; reprinted Bibliopolis, Naples, 1986.

[13] P. Clote. Sequential, machine independendent characterizations of the parallel complexity classes alogtime, ac^k and nc. In P. J. Scott and S. R. Buss, editors, *Feasible Mathematics*, pages 49–69. Birkhäuser, 1990.

[14] P. Clote. A safe recursion scheme for exponential time. In S. Adian and A. Nerode, editors, *Proceedings of Logical Foundations of Computer Science '97*, volume 1234 of *Lecture Notes in Computer Science*, pages 44–52. Springer, 1997.

[15] P. Clote. Computation models and function algebras. In E. R. Griffor, editor, *Handbook of Computability Theory*. Elsevier, Amsterdam, 1999.

[16] A. Cobham. The intrinsic computational difficulty of functions. In Y. B. Hillel, editor, *Proc.Int.Conf.Logic,Methodology and Philosophy Sci.*, pages 24–30. North Holland, Amsterdam, 1965.

[17] R. L. Constable. Classical proofs as programs. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, Series F: Computer and Systems Sciences, Volume 94, pages 31–61. Springer, Berlin, 1993.

[18] S. A. Cook and B. M. Kapron. Characterizations of the feasible functionals of finite type. In P. J. Scott and S. R. Buss, editors, *Feasible Mathematics*, pages 71–98. Birkhäuser, 1990.

[19] M. D. Davis and E. J. Weyker. *Computability, complexity and languages. Fundamentals of theoretical computer science*. Acad. Pr., New York, 1983.

[20] H. Friedman. Classically and intuitionistically provably recursive functions. In D. S. Scott and G. H. Müller, editors, *Higher Set Theory*, volume 669 of *Lecture Notes in Mathematics*, pages 21–28, Berlin, 1978. Springer.

[21] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[22] B. Goetze and W. Nehrlich. The structure of loop programs and subrecursive hierarchies. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 26, 1980.

[23] A. Grzegorczyk. Some classes of recursive functions. *Rozprawy Matematyczne*, (IV):1–45, 1953.

[24] P. Hajek and P. Pudlák. *Metamathematics of First-Order Arithmetic*. Ω Perspectives in Mathematical Logic. Springer-Verlag, Berlin Heidelberg, 1993.

[25] W. Heinermann. *Untersuchungen über die Rekursionszahlen rekursiver Funktionen*. PhD thesis, Münster, 1961.

[26] D. Hilbert. Über das Unendliche. *Mathematische Annalen*, 95:161–190, 1925.

[27] M. Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *The 1997 Annual Conference of the European Association for Computer Science Logic*, 1997.

[28] M. Hofmann. *Type systems for polynomial-time computation*. PhD thesis, TU Darmstadt, 1998. Habilitation Thesis.

[29] F. Joachimski and R. Matthes. Short proofs of normalisation for the simply-typed λ-calculus, permutative conversions and gödel's t. Submitted, 1998.

[30] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. In A. Heyting, editor, *Constructivity in Mathematics*, pages 101–128. North Holland, Amsterdam, 1959.

[31] L. Kristiansen and K.-H. Niggl. On the computational complexity of imperative programming languages. *Theoretical Computer Science*. To appear.

[32] D. Leivant. Subrecursion and lambda representation over free algebras. In S. Buss and P. Scott, editors, *Feasible Mathematics, Perspectives in Computer Science*, pages 281–291. Birkhäuser-Boston, New York, 1990.

[33] D. Leivant. A foundational delineation of computational feasibility. In *Proceedings of the Sixth IEEE Conference on Logic in Computer Science (Amsterdam)*, Washington D.C., 1991. IEEE Computer Society Press.

[34] D. Leivant. Stratified functional programs and computational complexity. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Langunages*, pages 325–333, New York, 1993. ACM.

[35] D. Leivant. Predicative recurrence in finite type. In A. Nerode and Y. V. Matiyasevisch, editors, *Logical Foundations of Computer Science*, volume 813, pages 227–239. Springer Lecture Notes in Computer Science, 1994.

[36] D. Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In P. Clote and J. Remmel, editors, *Feasible Mathematics II, Perspectives in Computer Science*, pages 321–343. Birkhäuser, 1994.

[37] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Fundamenta Informaticae*, 19(1,2):167–184, September 1993.

[38] D. Leivant and J.-Y. Marion. Ramified recurrence and computational complexity iv: Predicative functionals and poly-space. *Information and Computation*, To appear.

[39] M. Machthey. Augmented loop languages and classes of computable functions. *J. Comp. and System Sciences*, 6:603–624, 1972.

[40] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proc. ACM Nat. Conf.*, pages 465–469, 1967.

[41] H. Müller. *Klassifizierungen der primitiv rekursiven Funktionen*. PhD thesis, Münster, 1974.

[42] M. H. A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.

[43] A. Nguyen. *A Formal System For Linear-Space Reasoning*. PhD thesis, M.Sc. Thesis, Department of Computer Science, University of Toronto, available as T.R. 330/96, 1993.

[44] K.-H. Niggl. Towards the computational complexity of $\mathcal{PR}^{\omega}$-terms. *Annals of Pure and Applied Logic*, 75:153–178, 1995.

[45] K.-H. Niggl. The μ-measure as a Tool for Classifying Computational complexity. *The Bulletin of Symbolic Logic*, 4(1):100–101, March 1998.

[46] K.-H. Niggl. The μ-Measure as a Tool for Classifying Computational Complexity. *Archive for Mathematical Logic*, 39:515–539, 2000.

[47] I. Oitavem. New recursive characterizations of the elementary functions and the functions computable in polynomial space. *Revista Mathematica de la Universidad Complutense de Madrid*, 10(1), 1997.

[48] I. Oitavem. *Classes of Computational Complexity: Implicit Characterizations – a Study in Mathematical Logic*. PhD thesis, Faculdade de Ciências da Universidade de Lisboa, 2001.

[49] C. Parsons. Hierarchies of primitive recursive functions. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 14:357–376, 1968.

[50] R. Péter. *Recursive Functions*. New York London: Academic Press, 1967.

[51] R. W. Ritchie. Classes of predictably computable functions. *Trans. A.M.S.*, 106:139–173, 1963.

[52] R. W. Ritchie. Classes of recursive functions based on Ackermann's functions. *Pacific J. Math.*, 15:1027–1044, 1965.

[53] H. E. Rose. *Subrecursion. Functions and hierarchies*. Clarendon Press, Oxford, 1984.

[54] H. Schwichtenberg. Rekursionszahlen und die Grzegorczyk–Hierarchie. *Archiv für mathematische Logik und Grundlagenforschung*, 12(1–2):85–97, 1969.

[55] H. Schwichtenberg. A normal form for natural deductions in a type theory with realizing terms. In V. M Abrusci and E. Casari, editors, *Atti del Congresso Logica e Filosofia della Scienza, oggi, San Gimignano, 7–11 dicembre 1983, Vol.1–Logica*, pages 95–138, Bologna, 1986. CLUEB.

[56] H. Schwichtenberg. Proofs as programs. In P. Aczel, H. Simmons, and S. S. Wainer, editors, *Proof Theory. A selection of papers from the Leeds Proof Theory Program 1990*, pages 81–113. Cambridge University Press, Cambridge, 1991.

[57] H. Schwichtenberg. Classifying recursive functions. In E. R. Griffor, editor, *Handbook of Computability Theory*, chapter 16, pages 533–586. Elsevier Science, Amsterdam, 1999.

[58] H. Schwichtenberg. Feasible programs from proofs. Draft, 2000.

[59] H. Simmons. The Realm of Primitive Recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.

[60] W. W. Tait. Intensional interpretation of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

[61] A. S. Troelstra, editor. *Metamathematical Investigations of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*. Springer, Berlin, 1973.

[62] A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 1996.

Index