Available online at www.sciencedirect.com
SCIENCE DIRECT®
Theoretical Computer Science 318 (2004) 139–161
Theoretical Computer Science
www.elsevier.com/locate/tcs

On the computational complexity of imperative programming languages

L. Kristiansen[a], K.-H. Niggl[b,*]

[a] *Oslo University College, Faculty of Engineering, Norway*
[b] *Technische Universität Ilmenau, Institut für Theoretische und Technische Informatik, Helmholtzplatz, 1 Ilmenau 98693, Germany*

Abstract

Two restricted imperative programming languages are considered: One is a slight modification of a loop language studied intensively in the literature, the other is a stack programming language over an arbitrary but fixed alphabet, supporting a suitable loop concept over stacks. The paper presents a purely syntactical method for analysing the impact of nesting loops on the running time. This gives rise to a uniform measure μ on both loop and stack programs, that is, a function that assigns to each such program P a natural number $\mu(\mathtt{P})$ computable from the syntax of P.

It is shown that stack programs of μ-measure n compute exactly those functions computed by a Turing machine whose running time lies in Grzegorczyk class $\mathscr{E}^{n+2}$. In particular, stack programs of μ-measure 0 compute precisely the polynomial-time computable functions.

Furthermore, it is shown that loop programs of μ-measure n compute exactly the functions in $\mathscr{E}^{n+2}$. In particular, loop programs of μ-measure 0 compute precisely the linear-space computable functions.

MSC: 03D10; 03D15; 03D20; 68Q15; 68Q25

Keywords: Implicit computational complexity; Imperative programming languages; Subrecursion theory; Grzegorczyk hierarchy; Polynomial-time computability

1. Introduction

We study two restricted imperative programming languages: One is a slight modification of a loop language studied intensively in the literature, e.g. [9,20,21]; the other

* Corresponding author.
E-mail addresses: larskri@iu.hio.no (L. Kristiansen), niggl@tu-ilmenau.de (K.-H. Niggl).
URL: http://eiche.theoinf.tu-ilmenau.de/∼niggl/

0304-3975/$ - see front matter
doi:10.1016/j.tcs.2003.10.016

is a stack programming language over an arbitrary but fixed alphabet Σ, supporting a suitable loop concept over stacks.

Loop programs are built from primitive instructions `suc(X)`, `nil(X)`, `pred(X)` by sequencing $\mathtt{P}_1;\mathtt{P}_2$ and loop statements `loop X [Q]` where `X` is a variable. Variables serve as registers, each holding a natural number which can be manipulated by running a loop program.

Stack programs too contain variables serving as stacks, each holding a word over Σ which can be manipulated by running a stack program. Stack programs are built from primitive instructions `push(a,X)` for $\mathtt{a}\in\Sigma$, `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 in the control stack while preserving its contents.

A key problem in implicit complexity is to analyse the impact of nesting loops on the running time or computational complexity. In particular, are there methods of extracting information from the syntax of stack programs so as to distinguish programs that run in polynomial time from those that run in (iterated) exponential time? And if so, is there a general rationale behind these methods that gives insight as to why some nesting of loops may cause a blow up in computational complexity, while others do not?

In this paper, we propose a purely syntactical method, called μ-measure, that assigns to each program `P` a natural number $\mu(\mathtt{P})$ computable from the syntax of `P`. Answering the first question above, it is shown that stack programs of μ-measure 0 compute precisely the polynomial-time computable functions. This is an instance of a more general result that answers the second question above: Stack programs of μ-measure n compute exactly the functions computable by a Turing machine whose running time lies in Grzegorczyk class $\mathscr{E}^{n+2}$.

In terms of loop programs, we show that programs of μ-measure n compute exactly the functions in $\mathscr{E}^{n+2}$. This improves the result of Meyer and Ritchie [21] where the nesting depth of "loop programs" is related to the Grzegorczyk hierarchy at and above level 3. As well, it improves other characterisations of the Grzegorczyk hierarchy at and above level 3 by Schwichtenberg [29], Müller [22] and Parsons [26] which are all based on the "nesting depth" of primitive recursions (cf. [11]).

In the following, the main ideas behind the measure μ are explained in terms of stack programs. To give a full picture of the operational semantics, 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` cannot be altered during an execution of the loop. To provide access to each symbol in `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

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

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`.

It is obvious that we have to nest loops to a certain depth in order to obtain programs of a certain high computational complexity. On the other hand, it is well known from daily programming that some programs with "high loop nesting depth" like, e.g.

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

run in polynomial time. 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 $\mathtt{Y}$ holds the word $w\mathtt{a}^{|v_1|\cdots|v_l|}$ after the execution of $\mathtt{P}_1$. So "high loop nesting depth" is a necessary condition for high computational complexity, but it is not a sufficient condition. In this paper, we give syntactical criteria that separate loops which may cause a blow up in computational complexity from those which do not. To outline the main ideas, consider the following two 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)].
```

Observe that both $\mathtt{P}_2$ and $\mathtt{P}_3$ have nesting depth 2, and they look quite similar. However, if w is initially stored in $\mathtt{X}$, then $\mathtt{Z}$ holds the word $a^{2^{|w|}}$ after $\mathtt{P}_2$ is executed, while $a^{|w|\cdot(|w|+1)}$ is stored in $\mathtt{Z}$ after the execution of $\mathtt{P}_3$. Thus, we see that $\mathtt{P}_3$ runs in polynomial time whereas $\mathtt{P}_2$ has exponential running time. The gist of the matter lies in a (control) *circle* contained inside the outermost loop in $\mathtt{P}_2$: Inside the loop governed by $\mathtt{X}$, first $\mathtt{Y}$ *controls* $\mathtt{Z}$ in that $\mathtt{Z}$ is updated via $\mathtt{push(a,Z)}$ inside a loop governed by $\mathtt{Y}$, and then $\mathtt{Z}$ *controls* $\mathtt{Y}$ in the same sense. In contrast, there is no such circle in $\mathtt{P}_3$. In fact, it turns out that those stack programs where each body of a loop statement is *circle-free* compute exactly the functions in FPTIME. All those programs will receive μ-measure 0. These ideas generalise uniformly to all levels of the Grzegorczyk hierarchy. To focus on the critical case where $\mathtt{P}$ is a loop $\mathtt{foreach\ X\ [Q]}$, assume that $\mu(\mathtt{Q})$ is already determined. Suppose that $\mathtt{Q}$ is a sequence $\mathtt{Q}_1;\ldots;\mathtt{Q}_l$, in which case $\mu(\mathtt{Q})$ is $\max\{\mu(\mathtt{Q}_1),\ldots,\mu(\mathtt{Q}_l)\}$. Then a blow up in running time can only occur if $\mathtt{Q}$ *has a top circle*, that is, $\mathtt{Q}$ has a circle with respect to a control variable $\mathtt{Y}$ of some component $\mathtt{Q}_i$ of maximal μ-measure $\mu(\mathtt{Q})$. In this case, we define $\mu(\mathtt{P}) := 1+\mu(\mathtt{Q})$. In all other cases for $\mathtt{Q}$, we define $\mu(\mathtt{P}) := \mu(\mathtt{Q})$. In fact, we will show that these loops do not cause a blow up in running time.

Adding that all primitive instructions receive μ-measure 0, one easily verifies for the examples above that $\mu(\mathtt{P}_1)=\mu(\mathtt{P}_3)=0$ whereas $\mu(\mathtt{P}_2)=1$.

The measure μ is convenient for various reasons: Firstly, it operates on restricted close-to-machine languages but supporting a clear control structure. Secondly, it can be extended to extensions of the programming language under consideration providing features supported by many high-level programming languages. Thirdly, the measure μ is conceptually simple, and it characterises computationally relevant complexity classes. Thus, it can help to ground the concepts of computational complexity by

providing a reference point other than the original resource-based concepts. Finally, 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 imperative programs like "counting nesting depth".

Nonetheless, there are, as expected, limitations to any such purely syntactical method like μ: There will always be programs with polynomial running time but of a measure >0.

This paper builds on recent work on *ramified analysis of recursion* by Bellantoni and Niggl [4], and Niggl [23]. There a purely syntactical method for analysing the impact of nesting recursions on computational complexity has been proposed, in the context of ordinary schemata-based definitions in [4], and in the context of lambda terms over ground-type variables in [23]. Ramified analysis of recursion characterises uniformly the Grzegorczyk hierarchy at and above the linear-space level when based on primitive recursion. One obtains the same hierarchy of classes, except with FPTIME at the first level, when primitive recursion is replaced with recursion on notation.

Various ramification concepts as initiated by Simmons [30], Leivant [13–15], Bellantoni and Cook [3] have led to resource-free, purely functional characterisations of many complexity classes, such as the polynomial-time computable functions [3,16,18], the linear-space computable functions [1,15,24], NC^1 and polylog space [6], NP and the polynomial-time hierarchy [2], the Kalmár-elementary functions [25], and the exponential time functions of linear growth [8], among many others.

Ramification concepts have also proved fruitful in characterising complexity classes by higher-type recursion, such as the Kalmár-elementary functions [17], polynomial space [19], and recently FPTIME [5,12].

2. Preliminaries

We assume only basic knowledge about subrecursion theory. Readers unfamiliar with these subjects are referred to [7,10,28]. We summarise some basic definitions and facts about the Grzegorczyk hierarchy from Rose's book.

For unary functions f, f^k denotes the *kth iterate of* f, that is, $f^0(x)=x$ and $f^{k+1}(x)=$
$f(f^k(x))$. The *principal functions* $E_1,E_2,E_3,\ldots$ defined by $E_1(x)=x^2+2$ and $E_{n+2}(x)=E^x_{n+1}(2)$, enjoy the following *monotonicity properties*: For all $n,x,t\in\mathbb{N}$ one has $x+1\leqslant E_{n+1}(x)\leqslant E_{n+1}(x+1)$, $E_{n+1}(x)\leqslant E_{n+2}(x)$ and $E^t_{n+1}(x)\leqslant E_{n+2}(x+t)$.

A function f is defined by *bounded recursion* from functions g,h,b if for all $\vec{x},y$, $f(\vec{x},0)=g(\vec{x})$, $f(\vec{x},y+1)=h(\vec{x},y,f(\vec{x}))$ and $f(\vec{x},y)\leqslant b(\vec{x},y)$.

The *nth Grzegorczyk class* $\mathscr{E}^n$, $n\geqslant 2$, 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.

By Ritchie [27] the class $\mathscr{E}^2$ characterises the class FLINSPACE of functions computable by a Turing machine in linear space; $\mathscr{E}^3$ characterises the Kalmár-

elementary functions (cf. [28]). Every $f \in \mathscr{E}^n$ satisfies $f(\vec{x}) \leqslant E_{n-1}^{m_f}(\max(\vec{x}))$ for a constant m_f. Thus, every function in $\mathscr{E}^2$ is bounded by a polynomial, and $E_n \notin \mathscr{E}^n$, showing that each $\mathscr{E}^n$ is a proper subset of $\mathscr{E}^{n+1}$. The union of all these classes is identical to the set of primitive recursive functions.

Functions $f_1, \ldots, f_k$ are defined by *simultaneous recursion* from $g_1, \ldots, g_k$ and $h_1, \ldots, h_k$ if $f_i(\vec{x}, 0) = g_i(\vec{x})$ and $f_i(\vec{x}, y+1) = h_i(\vec{x}, y, f_1(\vec{x}, y), \ldots, f_k(\vec{x}, y))$ for $i = 1, \ldots, k$. If in addition each f_i is *bounded* by a function b_i, that is, $f_i(\vec{x}, y) \leqslant b_i(\vec{x}, y)$ for all $\vec{x}, y$, then f is defined by *bounded simultaneous recursion* from $g_1, \ldots, g_k, h_1, \ldots, h_k$, $b_1, \ldots, b_k$.

While each class $\mathscr{E}^{n+2}$ is closed under bounded simultaneous recursion, one application of unbounded simultaneous recursion from functions in $\mathscr{E}^{n+2}$ yields functions in $\mathscr{E}^{n+3}$.

3. Stack programs

In this section, we presuppose an arbitrary but fixed alphabet $\Sigma := \{\mathtt{a}_1, \ldots, \mathtt{a}_l\}$. We will define a stack programming language over Σ where programs are built from basic instructions `push(a,X)`, `pop(X)`, `nil(X)` by sequencing, conditional and loop statements. We assume an infinite supply of variables `X`, `Y`, `Z`, `O`, `U`, `V`, possibly with subscripts. Intuitively, variables serve as *stacks*, each holding a word over Σ which can be manipulated by running a stack program.

Definition 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 $\mathtt{P}_1, \mathtt{P}_2$ are stack programs, then so is the *sequence* $\mathtt{P}_1\,;\mathtt{P}_2$;
- if `P` is a stack program, then so is the *conditional* `if top(X)` $\equiv$ `a[P]`;
- if `P` is a stack program, then so is the *loop* `foreach X [P]`, provided that no imperative `push(a,X)`, `pop(X)` or `nil(X)` occurs in `P`.

We use $\mathscr{V}(\mathtt{P})$ to denote the set of variables occurring in `P`.

Note 1: Every stack program can be written uniquely in the form $\mathtt{P}_1\,; \ldots ;\mathtt{P}_k$ such that each component $\mathtt{P}_i$ 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 $\{A\}\,\mathtt{P}\,\{B\}$ to specify or reason about programs, the meaning being that if condition A is fulfilled before `P` is executed, then condition B is satisfied after the execution of `P`. For example, $\{\vec{\mathtt{X}} = \vec{w}\}\,\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{w}'\}$ reads *if the words* $\vec{w}$ *are stored in the stacks* $\vec{\mathtt{X}}$, respectively, before the execution of `P`, then $\vec{w}'$ are stored in $\vec{\mathtt{X}}$ after the execution of `P`. Similarly, $\{\vec{\mathtt{X}} = \vec{w}\}\,\mathtt{P}\,\{|\mathtt{X}_1| \leqslant f_1(|\vec{w}|), \ldots, |\mathtt{X}_n| \leqslant f_n(|\vec{w}|)\}$ reads *if the words* $\vec{w}$ *are stored in the stacks* $\vec{\mathtt{X}}$*, respectively, before the execution of* `P`*, then each word stored in* $\mathtt{X}_i$ *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 2 (Operational semantics). Imperatives, conditionals and loop statements in $\mathtt{P}_1;\ldots;\mathtt{P}_k$ 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)` $\equiv$ `a[P]` executes the body `P` if the top symbol on stack `X` is identical to letter `a`, otherwise the conditional statement is ignored.

Loop statements `foreach X [P]` are executed *call-by-value*, that is, first a *local copy* `U` of `X` is allocated and `P` is altered to `P'` by simultaneously replacing each "free occurrence"[1] of `X` in `P` (appearing as `if top(X)` $\equiv$ `a[Q]`) with `U`. Then the sequence

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

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

We say that a stack program `P` *computes* a function $f:(\Sigma^*)^n \to \Sigma^*$ if `P` has an *output variable* `O` and *input variables* $\mathtt{X}_{i_1},\ldots,\mathtt{X}_{i_l}$ 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 $\mathtt{X}_{i_1},\ldots,\mathtt{X}_{i_l}$.

A *subprogram* of another program is any substring which itself is a program.

4. The measure μ on stack programs

In the analysis of computational complexity of stack programs `P`, the interplay of two kinds of variables will play a major role: the sets $\mathscr{U}(\mathtt{P})$ and $\mathscr{C}(\mathtt{P})$. The former consists of all variables `X` which occur as `push(a,X)` in `P` and thus might be updated in a run of `P`, while the latter consists of all `X` controlling a non-trivial loop statement in `P`. Of course, in the presence of sequence statements these two sets need not be disjoint. More precisely, we define

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

Definition 3 (Control). For stack programs `P`, the *control in* `P`, denoted $\xrightarrow{\mathtt{P}}$, is defined as the transitive closure of the following binary relation $\prec_{\mathtt{P}}$ on $\mathscr{V}(\mathtt{P})$:

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

[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`.

In other words, X *controls* Y *in* P, that is, $\mathtt{X} \xrightarrow{\mathtt{P}} \mathtt{Y}$, if there exists a sequence of variables $\mathtt{X} \equiv \mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_l \equiv \mathtt{Y}$ such that $\mathtt{X}_1 \prec_{\mathtt{P}} \mathtt{X}_2 \prec_{\mathtt{P}} \cdots \prec_{\mathtt{P}} \mathtt{X}_{l-1} \prec_{\mathtt{P}} \mathtt{X}_l$.

Definition 4 (Measure μ). 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.
- $\mu(\mathtt{if\ top(X)} \equiv \mathtt{a[Q]}) := \mu(\mathtt{Q})$
- $\mu(\mathtt{Q}_1\mathtt{;\ Q}_2) := \max\{\mu(\mathtt{Q}_1), \mu(\mathtt{Q}_2)\}$
- 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 $\mathtt{Q} \equiv \mathtt{Q}_1\mathtt{;} \ldots \mathtt{;Q}_l$ is said to have 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 $\mathtt{Q}_1\mathtt{;} \ldots \mathtt{;Q}_{i-1}\mathtt{;\ Q}_{i+1}\mathtt{;} \ldots \mathtt{;Q}_l$.

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

As pointed out above, we will show that stack programs where each body of a loop is *circle-free*, that is, has no top circle, exactly compute the polynomial-time computable functions.

5. 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* $b \in \mathscr{E}^{n+2}$, that is, $|f(\vec{w})| \leqslant b(|\vec{w}|)$ 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 every function computed by a core program of μ-measure 0 has a polynomial length bound. 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 $\mathtt{P}'$, called *flattened out*, will be such that a straightforward inductive argument shows that every function computed by $\mathtt{P}'$ has a length bound in $\mathscr{E}^{n+2}$.

Definition 5 (Core programs). *Core programs* are stack programs built from imperatives `push(a,X)` by sequencing and loop statements.

Note 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 $\{\vec{\mathtt{X}} = \vec{w}\}\,\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{u}\}$ implies $|\vec{w}| \leqslant |\vec{u}|$ (component-wise), and if $|\vec{w}| \leqslant |\vec{w}'|$, $\{\vec{\mathtt{X}} = \vec{w}\}$ $\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{u}\}$ and $\{\vec{\mathtt{X}} = \vec{w}'\}\,\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{u}'\}$, then $|\vec{u}| \leqslant |\vec{u}'|$. Thus, functions computable by core programs are length-monotonic, too.

Lemma 6 (Measure zero). *For every core program* $\mathtt{P} :\equiv \mathtt{foreach\ X\ [Q]}$ *of* μ*-measure* 0, $\overset{\mathtt{P}}{\rightarrow}$ *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 $\mathtt{Q}$ is an imperative $\mathtt{push(a,Y)}$. If $\mathtt{Q}$ is of the form $\mathtt{foreach\ Y\ [R]}$, the statement follows from the induction hypothesis on $\mathtt{Q}$ and $\mathtt{X} \notin \mathscr{U}(\mathtt{Q})$. Finally, if $\mathtt{Q}$ is a sequence $\mathtt{Q}_1; \ldots; \mathtt{Q}_n$, then by the induction hypothesis on each component $\mathtt{Q}_i$, no $\mathtt{Y}$ controls $\mathtt{Y}$ in $\mathtt{Q}_i$. Therefore, if some $\mathtt{Y}$ controlled $\mathtt{Y}$ in $\mathtt{Q}$, then $\mathtt{Y}$ controlled some $\mathtt{Z} \not\equiv \mathtt{Y}$ in some $\mathtt{Q}_j$, and $\mathtt{Z}$ controlled $\mathtt{Y}$ in the context $\mathtt{Q}_1; \ldots; \mathtt{Q}_{j-1};\ \mathtt{Q}_{j+1}; \ldots; \mathtt{Q}_n$. Hence, $\mathtt{Q}$ would have a top circle, contradicting the hypothesis $\mu(\mathtt{P}) = 0$. □

Lemma 7 (Irreflexive bounding). *Let* $\mathtt{P}$ *be a core program of irreflexive* $\overset{\mathtt{P}}{\rightarrow}$. *Let* $\mathtt{P}$ *have variables among* $\vec{\mathtt{X}} := \mathtt{X}_1, \ldots, \mathtt{X}_n$, *and for* $i = 1, \ldots, n$ *let* V^i *denote the list of those variables* $\mathtt{X}_j$ *which control* $\mathtt{X}_i$ *in* $\mathtt{P}$. *Then there are polynomials* $p_1(V^1), \ldots, p_n(V^n)$ *such that for all* $\vec{w} := w_1, \ldots, w_n$,

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

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

Proof. By induction on the structure of core programs $\mathtt{P}$ with irreflexive $\overset{\mathtt{P}}{\rightarrow}$. In the *base case* $\mathtt{P} \equiv \mathtt{push(a,X_1)}$, we know $V^1 = \emptyset$ and hence $p_1 := 1$ will do.

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

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

Step case: $\mathtt{P} \equiv \mathtt{foreach\ X_j\ [Q]}$. The induction hypothesis yields polynomials $p_1(V^1), \ldots, p_n(V^n)$ such that for all $\vec{w}$,

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

As $\overset{\mathtt{P}}{\rightarrow}$ is irreflexive, then so is $\overset{\mathtt{Q}}{\rightarrow}$, implying that $\overset{\mathtt{Q}}{\rightarrow}$ defines a strict partial order on $\vec{\mathtt{X}}$. Thus, we can proceed by side induction on this partial order showing that one can find polynomials $q_1(m, V^1), \ldots, q_n(m, V^n)$ such that for all $m, \vec{w}$

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

where $\mathtt{Q}^m$ denotes the sequence $\mathtt{Q}; \ldots; \mathtt{Q}$ (m times $\mathtt{Q}$). Note that (2) implies the statement of the lemma for the current case $\mathtt{P} \equiv \mathtt{foreach\ X_j\ [Q]}$. To see this, if $\mathtt{X}_i \notin \mathscr{U}(\mathtt{Q})$ then V^i is empty, and the execution of $\mathtt{Q}$ does not alter the contents of $\mathtt{X}_i$, hence $p_i := 0$ will do. Otherwise if $\mathtt{X}_i \in \mathscr{U}(\mathtt{Q})$, then $\mathtt{X}_j \in V^i$ and (2) gives $\{\vec{\mathtt{X}} = \vec{w}\}\, \mathtt{P}\, \{|\mathtt{X}_i| \leqslant |w_i| + q_i(|w_j|, |\vec{w}^i|)\}$, where $w_j \in \vec{w}^i$.

As for the proof of (2), fix any $i \in \{1,\ldots,n\}$. If V^i is empty, then (1) implies that p_i is a constant, and hence $\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{|\mathtt{X}_i| \leqslant |w_i| + m \cdot p_i\}$. So consider the case where V^i is $\mathtt{X}_{i_1},\ldots,\mathtt{X}_{i_l}$ with $l \geqslant 1$. 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 $m, \vec{w}$,

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

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

$$\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{|\mathtt{X}_i| \leqslant |w_i| + m \cdot p_i(\ldots, |w_{i_j}| + r_{i_j}(m, |\vec{w}^{i_j}|),\ldots)\}. \tag{4}$$

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

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

The various induction hypotheses at hand yield the following estimations:

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

Combining these estimations and using monotonicity of polynomials, the required estimation $|v_i^*| \leqslant |w_i| + (m+1) \cdot p_i(\ldots, |w_{i_j}| + r_{i_j}(m+1, |\vec{w}^{i_j}|),\ldots)$ follows. This concludes the proof of (4) and thus that of the lemma. □

Corollary 8 (Base bounding). *For every core program* $\mathtt{P}$ *of* μ*-measure* 0 *and variables* $\vec{\mathtt{X}} := \mathtt{X}_1,\ldots,\mathtt{X}_n$ *one can find polynomials* $p_1(\vec{\mathtt{X}}),\ldots,p_n(\vec{\mathtt{X}})$ *such that for all* $\vec{w} := w_1,\ldots,w_n$

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

In particular, every function f *computed by* $\mathtt{P}$ *has a* polynomial length bound, *that is, a polynomial* p *satisfying* $|f(\vec{w})| \leqslant p(|\vec{w}|)$ *for all* $\vec{w}$.

Proof. The statement of the corollary follows from Irreflexive bounding (7), Measure zero (6), Note 1, and closure of polynomials under composition. □

To treat the general case in the proof of the Bounding Theorem, 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.

Definition 9 (Length bound). For stack programs $\mathtt{P}, \mathtt{Q}$ such that $\mathscr{V}(\mathtt{P}) = \vec{\mathtt{X}}$ and $\mathscr{V}(\mathtt{Q}) = \vec{\mathtt{X}}, \vec{\mathtt{Y}}$, we say that $\mathtt{Q}$ is a *length bound* on $\mathtt{P}$, denoted $\mathtt{P} \ll \mathtt{Q}$, if

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

For the proof of Flattening (14) below, fix any $n \in \mathbb{N}$.

Definition 10 (Flattened out). A loop `foreach X [Q]` of μ-measure $n+1$ is *simple* if $\mu(\mathtt{Q}) = n$. A core program $\mathtt{P} :\equiv \mathtt{P}_1\,;\cdots;\mathtt{P}_k$ of μ-measure $n+1$ is called *flattened out* if each component $\mathtt{P}_i$ is either a simple loop or else $\mu(\mathtt{P}_i) \leqslant n$.

Given a core program P with $\mu(\mathtt{P}) = n+1$, we want to construct a flattened out core program $\mathtt{P}'$ of the same μ-measure such that $\mathtt{P}'$ is a length bound on P. To that end, it suffices to transform, step by step, certain occurrences of non-simple loops in P. That motivates the definition of "rank" below where we make use of the standard notion of *nesting depth* deg, that is, $\deg(\mathtt{push(a,X)}) := 0$, $\deg(\mathtt{P}_1\,;\mathtt{P}_2) := \max\{\deg(\mathtt{P}_1), \deg(\mathtt{P}_2)\}$, and $\deg(\mathtt{foreach\ X\ [Q]}) := 1 + \deg(\mathtt{Q})$.

Definition 11 (Rank). The *rank* $\mathrm{rk}(\mathtt{P})$ of a core program P is inductively defined as follows:

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

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

Lemma 12 (Rank zero). *Every core program* P *with* μ*-measure* $n+1$ *and rank* 0 *is flattened out.*

Proof. By induction on the structure of core programs P with μ-measure $n+1$ and rank 0. If P is a sequence $\mathtt{P}_1\,;\mathtt{P}_2$ then both components are of rank 0, and at least one has μ-measure $n+1$. Hence, the claim follows from the induction hypothesis on those components with μ-measure $n+1$. If P is a loop, this loop is simple by definition, hence P is flattened out. □

Lemma 13 (Rank reduction). *For every core program* $\mathtt{P} :\equiv \mathtt{foreach\ X\ [Q]}$ *with* $\mu(\mathtt{P}) = n+1$ *and* $\mathrm{rk}(\mathtt{P}) > 0$, *one can find a core program* $\mathtt{P}'$ *satisfying* $\mathtt{P} \ll \mathtt{P}'$, $\mu(\mathtt{P}') = n+1$ *and* $\mathrm{rk}(\mathtt{P}') < \mathrm{rk}(\mathtt{P})$.

Proof. Let $\mathtt{P} :\equiv \mathtt{foreach\ X\ [Q]}$ be any core program with μ-measure $n+1$ and rank >0. According to Definition 11 and Note 1, we distinguish two cases.

Case: Q is a loop `foreach Y [R]` with $\mu(\mathtt{P}) = \mu(\mathtt{Q}) = n+1$. In this case, for some *new* variable Z and any letter a, we define $\mathtt{P}'$ by

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

Obviously, $\mu(\mathtt{P}') = \mu(\mathtt{P})$ and $\mathtt{P} \ll \mathtt{P}'$. As for $\mathrm{rk}(\mathtt{P}') < \mathrm{rk}(\mathtt{P})$, first observe that $\mathrm{rk}(\mathtt{P}) = 1 + \mathrm{rk}(\mathtt{Q})$ and $\mathrm{rk}(\mathtt{P}') = \mathrm{rk}(\mathtt{foreach\ Z\ [R]})$. Thus, we obtain $\mathrm{rk}(\mathtt{P}) > \mathrm{rk}(\mathtt{Q}) = \mathrm{rk}(\mathtt{foreach\ Y[R]}) = \mathrm{rk}(\mathtt{foreach\ Z\ [R]}) = \mathrm{rk}(\mathtt{P}')$ as required.

Case: Q is a sequence $\mathtt{Q}_0;\ldots;\mathtt{Q}_k$ without top circle, and $\mu(\mathtt{Q}_i)=\mu(\mathtt{Q})=n+1$ for some component $\mathtt{Q}_i :\equiv \mathtt{foreach\ Y\ [R]}$. In this case, the required $\mathtt{P}'$ is defined by $\mathtt{P}' :\equiv \mathtt{foreach\ X\ [P_1]\ ;\ foreach\ Z\ [P_2]}$ where for some *new* variable Z and any letter a the core programs $\mathtt{P}_1, \mathtt{P}_2$ are defined as follows:

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

As for $\mathtt{P}\ll\mathtt{P}'$, let $\#\mathtt{R}[\mathtt{P}(\vec{w})]$ denote the number of times R is executed in a run of P on $\vec{w}$. We conclude $\#\mathtt{R}[\mathtt{P}(\vec{w})]\leqslant\#\mathtt{push(a,Z)}[\mathtt{P}'(\vec{w},v)]$, for otherwise some $\mathtt{V}\in\mathscr{U}(\mathtt{R})$ controlled Y in $\mathtt{Q}_0;\ldots;\mathtt{Q}_{i-1};\mathtt{Q}_{i+1};\ldots;\mathtt{Q}_k$, contradicting the hypothesis on Q. Thus, $\{\vec{\mathtt{X}}=\vec{w},\mathtt{Z}=v\}\ \mathtt{foreach\ X\ [P_1]}\ \{|\mathtt{Z}|\geqslant\#\mathtt{R}[\mathtt{P}(\vec{w})]\}$, and that implies $\mathtt{P}\ll\mathtt{P}'$ by monotonicity of core programs.

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

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

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

Lemma 14 (Flattening). *For every core program* P *with* $\mu(\mathtt{P})=n+1$ *one can find a flattened out core program* $\mathtt{P}'$ *satisfying* $\mu(\mathtt{P}')=n+1$ *and* $\mathtt{P}\ll\mathtt{P}'$.

Proof. The statement of the lemma follows from Note 1, Rank reduction (13) and Rank zero (12). □

As pointed out above, Flattening (14) establishes the Bounding Theorem.

Theorem 15 (Bounding). *Every function f computed by a stack program of μ-measure n has a* length bound $b\in\mathscr{E}^{n+2}$, *i.e.* $|f(\vec{w})|\leqslant b(|\vec{w}|)$ *for all* $\vec{w}$.

Proof. It suffices to prove the statement of the theorem for core programs only, since 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 conditionals if top(X) $\equiv$ a[Q] with foreach X [push(b,V)]; $\mathtt{Q}^*$, for some *new* variable V and any letter b.

We proceed by induction on n showing the statement of the theorem for core programs. The *base case* has been shown in Corollary 8. As for the *step case* $n \to n+1$, let P be an arbitrary core program with μ-measure $n+1$. We apply Flattening (14) to obtain a core program $\mathtt{P}'$ of the form $\mathtt{P}_1; \ldots; \mathtt{P}_k$ where each component $\mathtt{P}_i$ is either a simple loop or else $\mu(\mathtt{P}_i) \leqslant n$, and such that $\mathtt{P} \ll \mathtt{P}'$ and $\mu(\mathtt{P}') = n+1$. Thus, by the induction hypothesis and by closure of $\mathscr{E}^{n+3}$ under composition, it suffices to show that every function computed by a simple loop $\mathtt{P}_i$ has a length bound in $\mathscr{E}^{n+3}$.

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 $b_j \in \mathscr{E}^{n+2}$. We choose a number $c > 0$ such that $b_j(\vec{x}) \leqslant E^c_{n+1}(\max(\vec{x}))$ 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 $|f_k(v, \vec{w})| \leqslant E^{c \cdot |v|}_{n+1}(\max(|v, \vec{w}|))$. As $E^t_{n+1}(x) \leqslant E_{n+2}(x+t)$ and $\max, + \in \mathscr{E}^2$, we thus obtain a length bound on f_1 in $\mathscr{E}^{n+3}$. □

6. The Characterisation theorem for stack programs

In this section, we will show that stack programs of μ-measure n compute precisely the functions computable by a Turing machine whose running time lies in $\mathscr{E}^{n+2}$. In particular, stack programs of μ-measure 0 compute exactly the functions in FPTIME.

Lemma 16 (E_{n+1}-Simulation). *Every E_{n+1} has a sequence* LE[n+1] *with a top circle satisfying* μ(LE[n+1]) $= n$ *and* $\{\mathtt{Y} = w\}$ LE[n+1] $\{|\mathtt{Y}| = E_{n+1}(|w|)\}$.

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 $E_{n+2}(x) = E_{n+1}(\ldots E_{n+1}(2)\ldots)$ 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 17 (Characterisation). *Stack programs of μ-measure n compute exactly the functions computable by a Turing machine whose running time lies in $\mathscr{E}^{n+2}$.*

Proof. First we treat the implication "⇒". So let P be any stack program of μ-measure n. Then let $\text{TIME}_{\mathtt{P}}(\vec{w})$ denote the number of steps in a run of P on input $\vec{w}$, where a *step* is the execution of any imperative imp(X). Observe that there is a polynomial $q_{\text{time}}(n)$ such that each step imp(X) can be simulated by a Turing machine in time $q_{\text{time}}(|\mathtt{X}|)$. Now let V be any *new* variable, and a any letter. Then let $\mathtt{P}^{\#}$ result from P by replacing each imperative imp with the sequence imp;push(a,V). We conclude that the program $\mathtt{TIME(P)} :\equiv \mathtt{nil(V)};\mathtt{P}^{\#}$ is of μ-measure n and satisfies $\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{TIME(P)}\ \{|\mathtt{V}| = \text{TIME}_{\mathtt{P}}(\vec{w})\}$. Thus, we may apply Bounding (15) to obtain a length bound $b \in \mathscr{E}^{n+2}$ satisfying

$$\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{TIME(P)}\ \{|\mathtt{V}| \leqslant b(|\vec{w}|)\}.$$

It follows that there exists a Turing machine which simulates P on input $\vec{w}$ in time $q_{\text{time}}(b(|\vec{w}|)) \cdot b(|\vec{w}|)$, concluding the proof of implication "⇒".

As for "⇐", let $M := (Q, \Gamma, \Sigma, q_0, B, F, \delta)$ be any one-tape Turing machine which on input w runs in time $b(|w|)$, for some $b \in \mathscr{E}^{n+2}$. The function f_M computed by M will be computed by a stack program P over $\Delta := Q \cup \Gamma \cup \{L, N, R\}$ of μ-measure n. Assume that δ consists of *moves* $\text{move}_1, \ldots, \text{move}_l$ where

$$\text{move}_i := (\mathtt{q}_i, \mathtt{a}_i, \mathtt{q}'_i, \mathtt{a}'_i, D_i) \quad \text{with } D_i \in \{L, N, R\}.$$

The required program P of μ-measure n satisfying $\{\mathtt{X} = w\}\ \mathtt{P}\ \{\mathtt{O} = f_M(w)\}$ uses stacks $\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{L}, \mathtt{R}, \ldots$, 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 be COMPUTE-TIME-BOUND(Y);INITIALISE(L,Z,R). The semantics of P is as expected: For each configuration $\alpha(\mathtt{q}, \mathtt{a})\beta$ obtained in a run of M on w after m steps, that is, $\text{init}_M(w) \vdash^m \alpha(\mathtt{q}, \mathtt{a})\beta$, we will have

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

Recall that $b(x) \leqslant E^c_{n+1}(x)$ for some constant c, and E_{n+1}-Simulation (16) yields a program of μ-measure n satisfying $\{\mathtt{Y} = w\}\ \mathtt{LE[n+1]}\ \{|\mathtt{Y}| = E_{n+1}(|w|)\}$. Thus, COMPUTE-TIME-BOUND(Y) can be defined by the following sequence:

$$\mathtt{nil(Y);\ foreach\ X\ [push(a,Y)];\ \ LE[n+1];\ldots;LE[n+1]}\quad (c \text{ times})$$

satisfying $\{\mathtt{X} = w\}$ COMPUTE-TIME-BOUND(Y) $\{\mathtt{X} = w,\ |\mathtt{Y}| = E^c_{n+1}(|w|)\}$. According to $(*)$, we initialise L, Z, R as follows:

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

where $\{\mathtt{X}=w\}$ `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` $\mathtt{X}\equiv\varepsilon$`[Q]` and `if` $\mathtt{X}\not\equiv\varepsilon$`[Q]` of μ-measure $\mu(\mathtt{Q})$ can be defined by

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

for some *new* variable `U`, and any letter `a`. Similarly, one defines conditionals `top(R)` $\in\Sigma$ `[Q]` and `if top(R)` $\in\Delta\backslash\Sigma$ `[Q]` of μ-measure $\mu(\mathtt{Q})$. We use

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

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

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

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 reverse(`R`) being a word over Σ. We define `OUTPUT(R;O)` as the following sequence:

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

This completes the proof of the Characterisation Theorem. □

It is worthwhile to point out that the proof of "⇐" 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 $\geqslant 2$ one cannot separate polynomial from exponential running time.

Furthermore, in the proof above, given a Turing machine M whose running time lies in $\mathscr{E}^{n+2}$, we have chosen a convenient alphabet (depending on M) in order to simulate

M by a stack program of μ-measure n. We stress that the use of arbitrary alphabets Σ has no influence on the μ-measure, for one can prove that every stack program over Σ can be simulated by a stack program over $\{0,1\}$ of the same μ-measure. Thus, we obtain that stack programs of μ-measure 0 compute exactly the polynomial-time computable functions.

7. Loop programs and the Grzegorczyk hierarchy

In this section, we will define another simple loop language. Programs written in this language are called *loop programs*. They are built from basic 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 $\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{O}, \mathtt{U}, \mathtt{V}$, 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 $\mathscr{E}^{n+2}$.

Definition 18 (Loop programs). *Loop programs* are inductively defined as follows:
- every *imperative* among `nil(X)`, `suc(X)`, `pred(X)` is a loop program;
- if $\mathtt{P}_1, \mathtt{P}_2$ are loop programs, then so is the *sequence* $\mathtt{P}_1\,;\mathtt{P}_2$;
- if `P` is a loop program, then so is the *loop* `loop X [P]`, provided that no imperative `suc(X)`, `nil(X)` or `pred(X)` occurs in `P`.

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

Note 3: Any loop program can be written uniquely in the form $\mathtt{P}_1\,;\ldots;\mathtt{P}_k$ such that each $\mathtt{P}_i$ 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 non-zero number stored in `X` by 1. 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, we define:

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

Now the concept of "control" for stack programs passes on to loop programs. As well, the measure μ on loop programs is defined as on stack programs, that is, $\mu(\mathtt{imp}) := 0$

for every imperative imp, $\mu(\mathtt{Q}_1\,;\ \mathtt{Q}_2) := \max\{\mu(\mathtt{Q}_1), \mu(\mathtt{Q}_2)\}$, and

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

where $\mathtt{Q} \equiv \mathtt{Q}_1\,; \ldots ;\mathtt{Q}_l$ is said to have 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 $\mathtt{Q}_1\,; \ldots ;\mathtt{Q}_{i-1}\,;\ \mathtt{Q}_{i+1}\,; \ldots ;\mathtt{Q}_l$.

Note that in the presence of nil(X), loop programs without pred(X) compute exactly the primitive recursive functions. In fact, pred(X) can be implemented by the following program PRED(X) satisfying $\{\mathtt{X} = x\}\,\mathtt{PRED(X)}\,\{\mathtt{Y} = x \dot{-} 1\}$:

PRED(X) :≡ nil(Y); nil(Z); loop X [nil(Y); loop Z [suc(Y)];suc(Z)].

Thus, modified subtraction $\dot{-}$ satisfying $x \dot{-} y = x - y$ for $x \geqslant y$, and 0 else, can be computed by loop U [PRED(X);nil(X); loop Y [suc(X)]]. Observe that this program is of μ-measure 1. It appears that without pred(X) as basic instruction there were no loop program of μ-measure 0 that computes modified subtraction. On the other hand, modified subtraction is a function in $\mathscr{E}^2$. So to obtain the characterisation above, we treat the predecessor function as a primitive instruction — as we could do with any non-increasing function in $\mathscr{E}^2$.

As with stack programs, the pivot in the proof of the characterisation theorem above 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 $\mathscr{E}^{n+2}$. No doubt, to establish this result, we could follow the course as we did to prove the Bounding Theorem (15) for stack programs. However, the way we choose is to benefit directly from Bounding (15).

Lemma 19 (Stack simulation). *For every loop program* P *using registers* $\vec{\mathtt{X}} := \mathtt{X}_1, \ldots, \mathtt{X}_l$ *one can find a stack program* $\mathscr{I}(\mathtt{P})$ *of the same* μ*-measure, using stacks* $\vec{\mathtt{X}}$ *over the unary alphabet* $\{\mathtt{a}\}$ *such that* $\{\vec{\mathtt{X}} = \vec{x}\}\,\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{y}\}$ *if and only if* $\{\mathtt{X}_1 = \mathtt{a}^{x_1}, \ldots, \mathtt{X}_l = \mathtt{a}^{x_l}\}$ $\mathscr{I}(\mathtt{P})\,\{\mathtt{X}_1 = \mathtt{a}^{y_1}, \ldots, \mathtt{X}_l = a^{y_l}\}$.

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

$$\mathscr{I}(\mathtt{nil(X)}) :\equiv \mathtt{nil(X)} \qquad \mathscr{I}(\mathtt{P}_1\,;\ \mathtt{P}_2) :\equiv \mathscr{I}(\mathtt{P}_1)\,;\ \mathscr{I}(\mathtt{P}_2),$$

$$\mathscr{I}(\mathtt{suc(X)}) :\equiv \mathtt{push}(a, \mathtt{X}) \quad \mathscr{I}(\mathtt{loop\ X\ [Q]}) :\equiv \mathtt{foreach\ X\ [}\mathscr{I}(\mathtt{Q})\mathtt{]},$$

$$\mathscr{I}(\mathtt{pred(X)}) :\equiv \mathtt{pop(X)}.$$

One can easily read off the definition that $\mathscr{I}(\mathtt{P})$ meets its specification. □

Theorem 20 (Loop bounding). *Every function* f *computed by a loop program of* μ*-measure* n *has a* bound $b \in \mathscr{E}^{n+2}$, *that is,* $f(\vec{x}) \leqslant b(\vec{x})$ *for all* $\vec{x}$.

Proof. Let P be any loop program of μ-measure n, and let f be any k-ary function computed by P. Consider the function $\mathscr{I}(f)$ computed by the stack program $\mathscr{I}(\mathtt{P})$

obtained from Stack simulation (19). Hence, $\mathscr{I}(\mathtt{P})$ is of μ-measure n, and $f(x_1,\ldots,x_k)=|\mathscr{I}(f)(\mathtt{a}^{x_1},\ldots,\mathtt{a}^{x_k})|$ for all $x_1,\ldots,x_k$. We apply Bounding (15) to obtain a length-bound $b\in\mathscr{E}^{n+2}$ on $\mathscr{I}(f)$, that is, $|\mathscr{I}(f)(\vec{w})|\leqslant b(|\vec{w}|)$ for all $\vec{w}$. Now, as $|\mathtt{a}^y|=y$, we conclude that b is a bound on f. □

Loop bounding (20) implies that every function computed by a loop program of μ-measure n lies in $\mathscr{E}^{n+2}$. To prove the converse, we adapt a technique developed in [23] (used in [4]) to our situation. It consists in separating the "structure" in a loop program from the "growth rate" given with it: Every $f\in\mathscr{E}^{n+2}$ can be *simulated* by a loop program `P[f]` of μ-measure 0 in the sense that `P[f]` on input $\vec{x},v$ outputs $f(\vec{x})$ whenever $v\geqslant E^m_{n+1}(\max(\vec{x}))$, 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]` is of μ-measure n and computes f. To that end, we need to implement appropriately "conditional statements".

Definition 21 (Conditional statements). For programs $\mathtt{P}:\equiv\mathtt{P}_1;\ldots;\mathtt{P}_k$ and variables `X`, `Y`, the *conditional statement* `if X⩽Y then [P]` is implemented as the sequence $\mathtt{Q}_1;\ \mathtt{Q}_2;\ \mathtt{Q}_3$ defined from *new variables* $\mathtt{U}^*,\mathtt{V}^*$ as follows:

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

Lemma 22 (Conditional). *Let* `P` *be any loop program, and let* $\mathtt{Z}_0,\ldots,\mathtt{Z}_n$ *be the variables of* $\mathscr{V}(\mathtt{P})\cup\{\mathtt{X},\mathtt{Y}\}$ *such that* $\mathtt{Z}_i=\mathtt{X}$ *and* $\mathtt{Z}_j=\mathtt{Y}$. *Then the program* $\mathtt{Q}:\equiv$ `if X⩽Y then [P]` *is of* μ*-measure* $\mu(\mathtt{P})$, *and*
(a) *if* $\{\vec{\mathtt{Z}}=\vec{z}\}\,\mathtt{P}\,\{\vec{\mathtt{Z}}=\vec{u}\}$, *then*

$$\begin{cases}\{\vec{\mathtt{Z}}=\vec{z}\}\,\mathtt{Q}\,\{\vec{\mathtt{Z}}=\vec{u}\} & \text{if } z_i\leqslant z_j,\\ \{\vec{\mathtt{Z}}=\vec{z}\}\,\mathtt{Q}\,\{\vec{\mathtt{Z}}=\vec{z}\} & \text{else},\end{cases}$$

(b) *for* $\mathtt{X}_0,\mathtt{X}_1\in\{\mathtt{Z}_0,\ldots,\mathtt{Z}_n\}$, $\mathtt{X}_0$ *controls* $\mathtt{X}_1$ *in* `Q` $\Leftrightarrow$ $\mathtt{X}_0$ *controls* $\mathtt{X}_1$ *in* `P`.

Proof. First, observe that $\mathtt{Q}_3$ could not be simply `loop V* [P]`, because that could result into $\mu(\mathtt{Q})>\mu(\mathtt{P})$. Hence $\mu(\mathtt{Q})=\mu(\mathtt{P})$ and (b) are obvious. As for part (a), assume that $\{\vec{\mathtt{Z}}=\vec{z}\}\,\mathtt{P}\,\{\vec{\mathtt{Z}}=\vec{u}\}$. As $\{\vec{\mathtt{Z}}=\vec{z}\}\,\mathtt{Q}_1\,\{\vec{\mathtt{Z}}=\vec{z},\mathtt{U}^*=z_i\dot{-}z_j\}$, we obtain $\{\vec{\mathtt{Z}}=\vec{z}\}\ \mathtt{Q}_1;\ \mathtt{Q}_2\,\{\vec{\mathtt{Z}}=\vec{z},\mathtt{V}^*=1\dot{-}(z_i\dot{-}z_j)\}$. This obviously implies (a), since $z_i\leqslant z_j\Leftrightarrow\mathtt{V}^*=1$, and $z_i>z_j\Leftrightarrow\mathtt{V}^*=0$. □

Lemma 23 (Loop simulation). *Let* `V` *be any variable. Every* $f\in\mathscr{E}^{n+2}$ *has a* simulation `P[f]` *and a* witness m_f *such that* $\mu(\mathtt{P[f]})=0$, $\mathtt{V}\notin\mathscr{U}(\mathtt{P[f]})$, *and*
(a) $f(\vec{x})\leqslant E^{m_f}_{n+1}(\max(\vec{x}))$ *for all* $\vec{x}$,
(b) $\{\vec{\mathtt{X}}=\vec{x},\mathtt{V}=v\}\,\mathtt{P[f]}\,\{\vec{\mathtt{X}}=\vec{x},\mathtt{V}=v,\mathtt{O}=f(\vec{x})\}$ *whenever* $v\geqslant E^{m_f}_{n+1}(\max(\vec{x}))$.

Proof. Induction on the structure of $f \in \mathscr{E}^{n+2}$. All cases are obvious where f is an initial function other than the principal function E_{n+1}. This case is reduced to the case of bounded recursion below, since E_{n+1} can be defined by bounded recursion from $E_n \in \mathscr{E}^{n+2}$ using E_{n+1} itself as bound.

Case: $f(\vec{x}) = h(g_1(\vec{x}), \ldots, g_k(\vec{x}))$ where $h, g_1, \ldots, g_k \in \mathscr{E}^{n+2}$. The induction hypothesis yields suitable simulations $\mathtt{P[h]}, \mathtt{P}[g_1], \ldots, \mathtt{P}[g_k]$ with witnesses $m_h, m_1, \ldots, m_k$, respectively. So for distinct variables $\mathtt{O}, \mathtt{O}_1, \ldots, \mathtt{O}_k$ we have $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = v\}\ \mathtt{P}[g_i]\ \{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = v, \mathtt{O}_i = g_i(\vec{x})\}$ whenever $v \geqslant E^{m_i}_{n+1}(\max(\vec{x}))$, and $\{\vec{O} = \vec{y}, \mathtt{V} = v\}\ \mathtt{P[h]}\ \{\vec{O} = \vec{y}, \mathtt{V} = v, \mathtt{O} = h(\vec{y})\}$ for $v \geqslant E^{m_h}_{n+1}(\max(\vec{y}))$. We define

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

Now, $\mu(\mathtt{P[f]}) = 0$ and (a) follow from the induction hypothesis. As for (b), consider any $\vec{x}, v$ with $v \geqslant E^{m_f}_{n+1}(\max(\vec{x}))$. Then we obtain $v \geqslant E^{m_i}_{n+1}(\max(\vec{x}))$ for $i = 1, \ldots, k$, and $v \geqslant E^{m_h}_{n+1}(\max(g_1(\vec{x}), \ldots, g_k(\vec{x})))$. Hence, (b) follows from (b) of the induction hypothesis.

Case: $f(\vec{x}, 0) = g(\vec{x})$, $f(\vec{x}, y+1) = h(\vec{x}, y, f(\vec{x}))$ and $f(\vec{x}, y) \leqslant b(\vec{x}, y)$ where $g, h, b \in \mathscr{E}^{n+2}$. The I.H. yields simulations $\mathtt{P[g]}, \mathtt{P[h]}$ with witnesses m_g, m_h. Thus, $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = v\}\ \mathtt{P[g]}\ \{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = v, \mathtt{O} = g(\vec{x})\}$ whenever $v \geqslant E^{m_g}_{n+1}(\max(\vec{x}))$, and $\{\vec{\mathtt{X}}, \mathtt{Y}, \mathtt{Z} = \vec{x}, y, z, \mathtt{V} = v\}\ \mathtt{P[h]}\ \{\vec{\mathtt{X}}, \mathtt{Y}, \mathtt{Z} = \vec{x}, y, z, \mathtt{V} = v, \mathtt{O} = h(\vec{x}, y, z)\}$ whenever $v \geqslant E^{m_h}_{n+1}(\max(\vec{x}, y, z))$. Now, choose m_b such that $b(\vec{x}, y) \leqslant E^{m_b}_{n+1}(\max(\vec{x}, y))$, and define the required witness m_f 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 possibly $\mu(\mathtt{P}) = 0$: For some *new* variable $\mathtt{L}$,

```
P :≡ P[g]; nil(L); loop Y [suc(L)]; nil(Y);
     loop L [nil(Z); loop O [suc(Z)]; P[h]; suc(Y)].
```

As for (b), assume that $v \geqslant E^{m_f}_{n+1}(\max(\vec{x}, y))$. Then $v \geqslant E^{m_g}_{n+1}(\max(\vec{x}))$, and $v \geqslant E^{m_h}_{n+1}(\max(\vec{x}, i, f(\vec{x}, i)))$ for $i < y$. The latter follows from $v \geqslant E^{m_f}_{n+1}(\max(\vec{x}, y)) \geqslant E^{m_h}_{n+1}(\max(\vec{x}, i, E^{m_b}_{n+1}(\max(\vec{x}, i)))) \geqslant E^{m_h}_{n+1}(\max(\vec{x}, i, f(\vec{x}, i)))$. Thus, we obtain $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = v\}\ \mathtt{P}\ \{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = v, \mathtt{O} = f(\vec{x})\}$. Now observe that $\mathtt{O}$ controls $\mathtt{Z}$ in $\mathtt{P}$, and $\mathtt{Z}$ is likely to control $\mathtt{O}$ in $\mathtt{P[h]}$, resulting into $\mu(\mathtt{P}) = 1$. To obtain the required simulation $\mathtt{P[f]}$, we modify $\mathtt{P}$ to $\mathtt{P[f]}$ as follows, using some *new* variable $\mathtt{U}$:

```
P[f] :≡ P[g]; nil(L); loop Y [suc(L)]; nil(Y);
        loop L [nil(Z); nil(U);
                loop V suc(U); if U ⩽ 0 then [suc(Z)]]; P[h];suc(Y)].
```

Let $\mathtt{R}$ be the subprogram `loop V [suc(U); if U⩽O then [suc(Z)]]`. Then, by unfolding the definition of the conditional in $\mathtt{R}$, we see that no variable from $\mathscr{C}(\mathtt{R})$ is controlled in $\mathtt{P[h]}$ by a variable from $\mathscr{U}(\mathtt{R})$, in particular $\mathtt{O} \notin \mathscr{C}(\mathtt{R})$, and that no variable from $\mathscr{C}(\mathtt{P[h]})$ is controlled in $\mathtt{R}$ by a variable from $\mathscr{U}(\mathtt{P[h]})$. Hence $\mu(\mathtt{P[f]}) = 0$. □

Lemma 24 (E_{n+1}-Computation). *Every E_{n+1} has a sequence* $\mathtt{E[n+1]}$ *with a top circle satisfying* $\mu(\mathtt{E[n+1]}) = n$ *and* $\{\mathtt{X} = x\}\ \mathtt{E[n+1]}\ \{\mathtt{X} = E_{n+1}(x)\}$.

Proof. By induction on n. The *base case* for $E_1(x) = x^2 + 2$ is obvious. As for the *step case*, 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, we therefore define

```
E[n+2] :≡ nil(Y); loop X [suc(Y)];
          nil(X); suc(X); suc(X); loop Y [E[n+1]].
```

□

Theorem 25 (Characterisation). *Loop programs of μ-measure n compute exactly the functions in $\mathscr{E}^{n+2}$.*

Proof. First consider any loop program P of μ-measure n. By Loop bounding (20), there is a function in $\mathscr{E}^{n+2}$ which is a bound on every function computed by a sub-program of P. Thus, we can proceed by induction on the structure of P showing that every function computed by P is in $\mathscr{E}^{n+2}$. All cases are obvious, except possibly the case where P is a loop. In this case, we make use of the fact that $\mathscr{E}^{n+2}$ is closed under bounded simultaneous recursion.

Conversely, consider any $f \in \mathscr{E}^{n+2}$. We apply Loop-simulation (23) to obtain P[f], m_f satisfying $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = v\}\ \mathtt{P[f]}\ \{\mathtt{O} = f(\vec{x})\}$ whenever $v \geqslant E_{n+1}^{m_f}(\max(\vec{x}))$. E_{n+1}-Computation (24) yields a program E[n+1] of μ-measure n such that $\{\mathtt{V} = y\}\ \mathtt{E[n+1]}\ \{\mathtt{V} = E_{n+1}(y)\}$. Using conditional statements one easily defines a loop program M of μ-measure 0 satisfying $\{\vec{\mathtt{X}} = \vec{x}\}\ \mathtt{M}\ \{\vec{\mathtt{X}} = \vec{x}, \mathtt{V} = \max(\vec{x})\}$. Thus, the sequence `M; E[n+1]; ...;E[n+1];P[f]`, with m_f occurrences of program E[n+1], defines a loop program of μ-measure n which computes f. □

Theorem 25 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 *running time* of loop programs, that is, for every loop program P of μ-measure n the function $\mathrm{TIME}_{\mathtt{P}}$ belongs to $\mathscr{E}^{n+2}$, where $\mathrm{TIME}_{\mathtt{P}}(\vec{x})$ 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 $\mathrm{TIME}_{\mathtt{P}}$, that is, $\{\vec{\mathtt{X}} = \vec{x}\}\ \mathtt{TIME(P)}\ \{\mathtt{V} = \mathrm{TIME}_{\mathtt{P}}(\vec{x})\}$. Hence $\mathrm{TIME}_{\mathtt{P}} \in \mathscr{E}^{n+2}$ by Theorem 25.

8. Sound, adequate and complete measures

We have presented a purely syntactical method for analysing the impact of nesting loops in imperative programs on the running time. In particular, the method separates programs which run in polynomial time (in the size of the input) from those which might run super-polynomial time. More generally, the method uniformly distinguishes programs whose running time lies in $\mathscr{E}^{n+2}$ from those whose running time might be beyond $\mathscr{E}^{n+2}$.

One might ask how successful this project can be, e.g. does every 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 26. Assume an arbitrary imperative programming language L and any program P in L. P is *feasible* if every function computed by P is in FPTIME. 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.

Observe 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 which runs in polynomial time. Dishonest programs fall into two groups. One group consists of those programs which only compute functions in FPTIME but with super-polynomial running time. The other group consists of programs which run in polynomial time, but some subprograms run in super-polynomial time if executed separately. A case in point is a program 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 any 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".

No doubt, we cannot expect to separate (by purely syntactical means) the feasible programs from the non-feasible ones if we allow for dishonest programs. Thus, it seems reasonable to consider only 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 27. Given any stack programming language L, a *measure on* L is a computable function $v : L\text{-programs} \to \mathbb{N}$.

Definition 28. Let L be any stack programming language containing core programs (cf. Section 5), and let v be any measure on L. The pair (v, L) is called
- *sound* if every L-program of v-measure 0 is feasible,
- *complete* if every honestly feasible L-program has v-measure 0, and
- *adequate* if every function in FPTIME is L-computable with v-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 29. *The pair* (μ, C) *is sound and complete.*

Proof. Soundness follows directly from Theorem 17. As for completeness, assume that P were an honestly feasible core program with $\mu(\mathrm{P}) > 0$. Then P contained a loop `foreach X [Q]`, where Q is a sequence $\mathrm{Q}_1; \ldots; \mathrm{Q}_l$ with a top circle. So Q

contained a component $\mathtt{Q}_i$ such that some Y controls some Z in $\mathtt{Q}_i$, and Z controls Y in $\mathtt{Q}_1;\ldots;\mathtt{Q}_{i-1};\ \mathtt{Q}_{i+1};\ldots;\mathtt{Q}_l$. 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. 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 on P. □

Clearly, (μ, C) is not adequate. As core programs are length-monotonic, there are plenty of functions in FPTIME which are not C-computable, let alone C-computable with μ-measure 0. However, would not 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 30. *Let (v, L) be sound and adequate. Then (v, L) is incomplete, that is, there exists an honestly feasible program $\mathtt{P} \in L$ such that $v(\mathtt{P}) > 0$.*

Proof. Assume an effective enumeration $\{\rho_i\}_{i\in\omega}$ of the Turing machines with alphabet $\{0,1\}$. Let n be a fixed natural number. It is well known that there is a function $f_n \in$ FPTIME satisfying $f_n(x) = 1$ if ρ_n (on the empty input) halts within $|x|$ steps, and $f_n(x) = 0$ else. Moreover, it is undecidable whether ρ_i halts. Since (v, L) is adequate and sound, there is an honestly feasible program Q in L of v-measure 0 such that

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

Such a program Q can be effectively constructed from n, i.e. there is an algorithm for constructing Q from n. As L contains the core language, the program

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

is also in L. 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 `foreach W [push(1,V)]` will be executed each time the body of the outermost loop is executed whenever $\mathtt{Y} = y$ with $|y| \geqslant s$. Each such execution at least doubles the height of V. Thus, if ρ_n 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 (v, L) cannot be complete. For if (v, L) were complete, then ρ_n would never halt if and only if $v(\mathtt{P}) = 0$. This would yield an algorithm which decides whether ρ_n halts: construct P from n and then check whether $v(\mathtt{P}) > 0$. Such an algorithm does not exist. □

Notably, Theorem 30 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 identifies the true statements of the language. Theorem 30 says that when a programming language is sufficiently expressive, then there is no algorithm which identifies the honestly feasible programs of the language.

Acknowledgements

We would like to express our gratitude to Martin Dietzfelbinger for numerous inspiring discussions and useful comments on various drafts of this paper. Also we would like to thank the referees for their thorough proofreading and useful suggestions which all helped to improve readability.

References

[1] S.J. Bellantoni, Predicative recursion and computational complexity, Ph.D. Thesis, Toronto, September 1993.
[2] S.J. Bellantoni, Predicative recursion and the polytime hierarchy, in: P. Clote, J. Remmel (Eds.), Feasible Mathematics II, Perspectives in Computer Science, Birkhäuser, Basel, 1994, pp. 15–29.
[3] S.J. Bellantoni, S. Cook, A new recursion-theoretic characterization of the polytime functions, Comput. Complexity 2 (1992) 97–110.
[4] S.J. Bellantoni, K.-H. Niggl, Ranking primitive recursions: the low Grzegorczyk classes revisited, SIAM J. Comput. 29 (2) (2000) 401–415.
[5] S.J. Bellantoni, K.-H. Niggl, H. Schwichtenberg, Higher type recursion, ramification and polynomial time, Ann. Pure Appl. Logic 104 (2000) 17–30.
[6] S. Bloch, Function-algebraic characterizations of log and polylog parallel time, Comput. Complexity 4 (2) (1994) 175–205.
[7] P. Clote, Computation models and function algebra, in: E. Griffor (Ed.), Handbook of Computability Theory, Elsevier, Amsterdam, 1996.
[8] P. Clote, A safe recursion scheme for exponential time, in: S. Adian, A. Nerode (Eds.), Lecture Notes in Computer Science, Vol. 1234, Springer, Berlin, 1997, pp. 44–52.
[9] B. Goetze, W. Nehrlich, The structure of loop programs and subrecursive hierarchies, Z. Math. Logik Grundlag. Math. 26 (1980) 255–278.
[10] A. Grzegorczyk, Some classes of recursive functions, Rozprawy Mat., Vol. IV, Warszawa, 1953.
[11] W. Heinermann, Untersuchungen über die Rekursionszahlen rekursiver Funktionen, Ph.D. Thesis, Münster, 1961.
[12] M. Hofmann, Type systems for polynomial-time computation, Habilitation Thesis, TU Darmstadt, 1998.
[13] D. Leivant, Subrecursion and lambda representation over free algebras, in: S. Buss, P. Scott (Eds.), Feasible Mathematics, Perspectives in Computer Science, Birkhäuser, Boston, New York, 1990, pp. 281–291.
[14] D. Leivant, A foundational delineation of computational feasibility, in: Proc. Sixth IEEE Conf. on Logic in Computer Science (Amsterdam), IEEE Computer Society Press, Washington, DC, 1991.
[15] D. Leivant, Stratified functional programs and computational complexity, in: Conf. Record of the 20th Annual ACM Symposium on Principles of Programming Languages, New York, 1993, pp. 325–333.
[16] D. Leivant, Ramified recurrence and computational complexity I: Word recurrence and poly-time, in: P. Clote, J. Remmel (Eds.), Feasible Mathematics II, Perspectives in Computer Science, Birkhäuser, Basel, 1994, pp. 320–343.
[17] D. Leivant, Predicative recurrence in finite type, in: A. Nerode, Y.V. Matiyasevisch (Eds.), Logical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 813, Springer, Berlin, 1994, pp. 227–239.
[18] D. Leivant, J.-Y. Marion, Lambda calculus characterizations of poly-time, Fund. Inform. 19 (1993) 167–184.
[19] D. Leivant, J.-Y. Marion, Ramified recurrence and computational complexity IV: predicative functionals and poly-space, Inform. Comput., to appear.
[20] M. Machthey, Augmented loop languages and classes of computable functions, J. Comput. System Sci. 6 (1972) 603–624.
[21] A.R. Meyer, D.M. Ritchie, The complexity of loop programs, in: Proc. ACM Nat. Conf., 1967, pp. 465–469.

[22] H. Müller, Klassifizierungen der primitiv rekursiven Funktionen, Ph.D. thesis, Münster, 1974.
[23] K.-H. Niggl, The μ-measure as a tool for classifying computational complexity, Arch. Math. Logic 39 (2000) 515–539.
[24] A. Nguyen, A formal system for linear-space reasoning, M.Sc. Thesis, University of Toronto, 1993; available as T.R. 330/96.
[25] I. Oitavem, New recursive characterizations of the elementary functions and the functions computable in polynomial space, Rev. Math. Univ. Complutense Madrid 10 (1) (1997) 109–125.
[26] C. Parsons, Hierarchies of primitive recursive functions, Z. Math. Logik Grundlag. Math. 14 (1968) 357–376.
[27] R.W. Ritchie, Classes of predictably computable functions, Trans. Amer. Math. Soc. 106 (1963) 139–173.
[28] H.E. Rose, Subrecursion, Functions and Hierarchies, Clarendon Press, Oxford, 1984.
[29] H. Schwichtenberg, Rekursionszahlen und die Grzegorczyk-Hierarchie, Arch. Math. Logik Grundlag. 12 (1969) 85–97.
[30] H. Simmons, The realm of primitive recursion, Arch. Math. Logic 27 (1988) 177–188.