Basic Research in Computer Science

The Implicit Computational Complexity of Imperative Programming Languages

Lars Kristiansen

BRICS Report Series **RS-01-46**

ISSN 0909-0878 **November 2001**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/01/46/`

The Implicit Computational Complexity of Imperative Programming Languages

Lars Kristiansen*.

Preface

This report presents research conceived during a few months the fall 2001 when I was visiting BRICS at the University of Århus. I would like emphasise the preliminary character of the report. So please forgive me all the inaccuracies, the unpolished presentation, the lack of references and acknowledgements, the sketchy proofs and the outright errors that might be there. The aim is to quickly communicate some ideas and results before I continue the work in a more comfortable and pleasant pace, either on my own or in cooperation with others.

I would like to thank The Faculty of Engineering at Oslo University College, BRICS, and in particular Ulrich Kohlenbach, for supporting my research. I would like to thank Ivar Rummelhoff for comments on the manuscript. Finally I would like to thank Karl-Heinz Niggl for numerous inspiring discussions.

Lars Kristiansen
Århus, November, 2001

*Oslo University College, Faculty of Engineering.
e-mail:larskri@iu.hio.no, home page:http://www.iu.hio.no/~larskri/
I dedicate this research to Jasmina Reza.

Abstract

During the last decade Cook, Bellantoni, Leivant and others have developed the theory of implicit computational complexity, i.e. the theory of predicative recursion, tiered definition schemes, etcetera. We extend and modify this theory to work in a context of imperative programming languages, and characterise complexity classes like P, LINSPACE, PSPACE and the classes in the Grzegorczyk hiearchy. Our theoretical framework seems promising with respect to applications in engineering.

1 Preliminaries

We assume some basic knowledge about subrecursion theory, in particular about the Grzegorczyk hierarchy. Readers unfamiliar with these subjects are referred to Grzegorczyk [9], Rose [29], Odifreddi [26] or Clote [6]. We summarise some basic definitions and facts from Rose.

For unary functions f, f^k denotes *kth iterate of* f, that is, $f^0(x) = x$ and $f^{k+1}(x) = f(f^k(x))$. The sequence $E_1, E_2, E_3, \ldots$ of number-theoretic functions is defined by $E_1(x) = x^2 + 2$ and $E_{n+2}(x) = E^x_{n+1}(2)$. We have the following monotonicity properties: $x + 1 \leq E_{n+1}(x)$, $E_{n+1}(x) \leq E_{n+1}(x + 1)$, $E_{n+1}(x) \leq E_{n+2}(x)$ and $E^t_{n+1}(x) \leq E_{n+2}(x + t)$ for all n, x, t.

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

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. The *0th Grzegorczyk class* $\mathcal{E}^0$ is the least class of functions containing the initial functions zero, successor, projections and is closed under composition and bounded recursion. The *1st Grzegorczyk class* $\mathcal{E}^1$ is the least class of functions containing the initial functions zero, successor, projections, addition and is closed under composition and bounded recursion.

By Ritchie [28] the class $\mathcal{E}^2$ characterises the class LINSPACEF of functions computable by a Turing machine in linear space; $\mathcal{E}^3$ equals the Kalmár-elementary functions (cf.[29]). Every $f \in \mathcal{E}^n$ satisfies $f(\vec{x}) \leq E^k_{n-1}(\max(\vec{x}))$ for a fixed number k. Thus, every function in $\mathcal{E}^2$ is bounded by a polynomial, and $E_n \notin \mathcal{E}^n$, showing that each $\mathcal{E}^n$ is a proper subset of $\mathcal{E}^{n+1}$. Every $f \in \mathcal{E}^0$ satisfy $f(x_1, \ldots, f_k) \leq x_i + k$ for some fixed numbers k and i (where $1 \leq i \leq k$), every $f \in \mathcal{E}^1$ satisfy $f(\vec{x}) \leq k \max(\vec{x}) + l$ for some fixed numbers k and l. Thus, we also have $\mathcal{E}^0 \subset \mathcal{E}^1 \subset \mathcal{E}^2$. The union of all the Grzegorczyk classes is identical to the set of primitive recursive functions.

If $\mathcal{F}$ is a class of functions, $\mathcal{F}_\star$ denotes the corespondent relational class, i.e.

$$\mathcal{F}_\star = \{f \mid f \in \mathcal{F} \text{ and } \mathrm{ran}(f) = \{0, 1\}\} .$$

The class $\mathcal{E}^{i+1}$ contains a universal function for the class $\mathcal{E}^i$ whenever $i \geq 2$. Thus, we have $\mathcal{E}^i_\star \subset \mathcal{E}^{i+1}_\star$ for $i \geq 2$. It is not known whether any of the inclusions $\mathcal{E}^0_\star \subseteq \mathcal{E}^1_\star \subseteq \mathcal{E}^2_\star$ are strict. See e.g. Rose [29] or Kutylowski [14].

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, $f_i(\vec{x}, y) \leq b_i(\vec{x}, y)$ for all $\vec{x}, y$, then f is said to be defined by *bounded simultaneous recursion* from $g_1, \ldots, g_k, h_1, \ldots, h_k, b_1, \ldots, b_k$. Note that for $i = 1$ the definition scheme for bounded simultaneous recursion degenerates to the scheme for bounded recursion. Hence every class closed under bounded simultaneous recursion is also closed under bounded recursion, but not necessarily vice versa.

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

We also assume some familiarity with complexity-theoretic classes like $\mathtt{P}$ (the class of problems decided by a Turing machine working in polynomial time in the length of input), LINSPACE (the class of problems decided by a Turing machine working in linear space in the length of input), PSPACE (the class of problems decided by a Turing machine working in polynomial space in the length of input), CONSPACE (the class of problems decided by a Turing machine which working in constant space). We use PF, LINSPACEF and PSPACEF denotes the correspondent classes of functions. On some occasions we view these classes as classes a of number theoretic functions, i.e. functions from tuples of natural numbers into the natural numbers, on other occasions we view these classes as classes of functions from tuples of words over an alphabet into the words of an alphabet. The readers not familiar with these classes should consult e.g. Odifreddi [26].

We will use informal Hoare-like sentences to specify or reason about imperative programs, that is, we will use the notation $\{A\}\,\mathtt{P}\,\{B\}$, the meaning being that if the condition given by the sentence A is fulfilled before $\mathtt{P}$ is executed, then the condition given by the sentence B is fulfilled after the execution of $\mathtt{P}$. For example, $\{\vec{\mathtt{X}} = \vec{w}\}\,\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{w}'\}$ reads as *if the data $\vec{w}$ are stored in the registers (or variables) $\vec{\mathtt{X}}$, respectively, before the execution of $\mathtt{P}$, then data $\vec{w}'$ are stored in $\vec{\mathtt{X}}$ after the execution of $\mathtt{P}$*. Another typical example is $\{\vec{\mathtt{X}} = \vec{w}\}\,\mathtt{P}\,\{|\mathtt{X}_1| \leq f_1(|\vec{w}|), \ldots, |\mathtt{X}_n| \leq f_n(|\vec{w}|)\}$ meaning that if the data $\vec{w}$ are stored in the registers $\vec{\mathtt{X}}$, respectively, before the execution of $\mathtt{P}$, then the data stored in X_i after the execution of $\mathtt{P}$ has a length bounded by $f_i(|\vec{w}|)$. Here f_i is function with range $\mathbb{N}$, and $|\vec{w}|$ abbreviates as usual the list $|w_1|, \ldots, |w_n|$. As usual $|x|$ denote the length of the data x.

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

A program $\mathtt{P}$ computes a functions f when $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\mathtt{Y} = f(\vec{x})\}$ for some $\vec{\mathtt{X}}, \mathtt{Y} \in \mathcal{V}(\mathtt{P})$. If $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\mathtt{Y} = f(\vec{x})\}$ we will also occasionally refer to f as *the function* $\mathtt{P}$ *computes into* $\mathtt{Y}$. This is a slightly sloppy use of words since $\mathtt{P}$ might very well compute many functions into $\mathtt{Y}$, e.g. if $\mathtt{Z} \in \mathcal{V}(\mathtt{P})$ and $\mathtt{Z} \notin \{\vec{\mathtt{X}}\}$ then we also have $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{Z} = z\}\mathtt{P}\{\mathtt{Y} = g(\vec{x}, z) = f(\vec{x})\}$ for some function g. When we construct programs, we occasionally need what we call *fresh* variables. That a variable is *fresh* simply means that the variable is not used elsewhere.

Let $\mathtt{P}$ and $\mathtt{Q}$ be imperative programs where the registers holds values of $\mathbb{N}$. Assume $(\mathtt{P}) \subseteq (\mathtt{Q})$, say $\mathcal{V}(\mathtt{P}) = \mathtt{X}$ and $\mathcal{V}(\mathtt{Q}) = \vec{\mathtt{X}}, \vec{\mathtt{Y}}$. We say that $\mathtt{Q}$ is a *bound* on $\mathtt{P}$, denoted $\mathtt{P} \ll \mathtt{Q}$, if

$$\{\vec{\mathtt{X}} = \vec{x}\}\ \mathtt{P}\ \{\vec{\mathtt{X}} = \vec{z}\} \text{ and } \{\vec{\mathtt{X}} = \vec{x}, \vec{\mathtt{Y}} = \vec{y}\}\ \mathtt{Q}\ \{\vec{\mathtt{X}} = \vec{u}\} \text{ implies } \vec{x} \leq \vec{u}.$$

2 Introduction

Implicit computational complexity theory can be viewed as theory on how to design programming languages that capture complexity classes. There are of course much more to implicit computational complexity than this, e.g. proof theoretical aspects, philosophical aspects, etcetera. We are talking about a broad and rich research area, but still, if we should explain the area to a layman or an engineer, it would not be too misleading to say that this is an area where we search for programming languages that captures complexity classes. The engineer would probably respond that this seems like a worthy endeavour, and then he[1] would expect us to come up with a nice practical programming language which he can use in his daily work. If we view the formalisms developed in Bellantoni-Cook [1], Leivant [17], and several other entries in the bibliography, as programming languages, we do not have exactly what an engineer conceives as practical programming languages. He would probably have problems to carry out simple tasks, like multiplying two numbers, in any of these formalisms.

What would be the most precious gift implicit computational complexity theory possibly could give to an engineer? Well, perhaps we could give him a neat practical programming language reminiscent of Pascal or C, a

[1] We assume that women do not exist.

language which permits him to write programs more or less in way he is used to, but still a language which endows him with means to easily retrieve useful information about the computational complexity of the programs. The language should be rich and able to express as many algorithms as possible, also algorithms of high computational complexity because sometimes the engineer indeed wants to implement such algorithms. (He makes sure that such programs only are executed on inputs they can cope with.) Will there ever be possible to provide such a language which engineers actually will use, either to develop programs that compiles into executable code on some electronic gadgets, or as a pen and paper language for analysing algorithms before they eventually are implemented by other means? As we will see, there are theoretical results and mathematical insights that put obstacles in the way of such a programming language, in spite of these results and insights we are able to present a theoretical framework which seems promising for developing languages satisfying the needs and requirements of engineering.

The main ideas are simple: We introduce programming languages capable of computing a huge class of functions e.g. every primitive recursive functions. We introduce measures on programs.

Definition 2.1. Given any programming language L, a *measure on* L is a computable function $\pi\colon L\text{-programs} \to \mathbb{N}$. We say that the program $\mathtt{P}$ has π-measure n if $\pi(\mathtt{P}) = n$. *End of definition.*

The idea is that a measure should tell us something about the computational complexity of a program, for example, if $\mathtt{P}$ has π-measure 0, then every function computed by $\mathtt{P}$ is in the complexity class $\mathcal{C}$ where $\mathcal{C}$ might be PF, LINSPACEF, the Kalmár elementary functions, or any other complexity class. One might ask how successful such a project can be. Is it for for example possible to find a nontrivial programming language and a measure π such that every program with polynomial running time (and only programs with polynomial running time) will receive π-measure 0? Unfortunately, the answer to this question is not an unqualified yes, it depends on what we mean by a nontrivial programming language. By ordinary computability-theoretic methods we shall prove some negative results. Thereafter we shall see that the situation is not as bad as these results may indicate at the first glance.

We start off by defining some generic languages.

Definition 2.2 *(Loop programs).* We have an infinite supply of program variables (registers). We will normally use $\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{U}, \mathtt{V}$ with or without

subscripts and superscripts to denote variables. For any variables `X` and `Y` we have the following *imperatives* (primitive instructions): `nil(X)`, `suc(X)`, `pred(X)`, `X:= Y`.

A *loop language* is a set of programs generated by the following syntactical rules:

- every imperative is a program
- if `P` is a program with no occurrence of the variable `X` in an imperative, then the *loop* `loop X [P]` is a program
- if $\mathtt{P}_1, \mathtt{P}_2$ are programs, then the *sequence* $\mathtt{P}_1\mathtt{;}\ \mathtt{P}_2$ is a program

L_0 is the loop language which only has the imperative `suc(X)` (for any variable `X`). L_1 is the loop language which has the imperatives `suc(X)` and `X:= Y` (for any variables `X` and `Y`). L is the loop language which has the imperatives `nil(X)`, `suc(X)`, `pred(X)` and `X:= Y` (for any variables `X` and `Y`).

Loop programs have a standard semantics, e.g. like Pascal or C programs. The imperative `nil(X)` sets the 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, `X:= Y` is ordinary assignment. 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`, besides, `X` keeps the value x during the execution of the loop body `P`. *End of definition.*

Example 2.3. If a variable `X` governs a loop, then `X` cannot occur in an imperative in the body of the loop. Still `X` is allowed to govern a loop in the body. The following L-program square the number stored in the register `X`.

$$\{\mathtt{X} = x\}\ \mathtt{nil(Y);\ loop\ X\ [\ loop\ X\ [\ suc(Y)\ ]\ ];\ X:=\ Y}\ \{\mathtt{X} = x^2\}$$

End of example.

Definition 2.4. Let `P` be a program in a loop language and let $\mathcal{C}$ be a complexity class. `P` is *$\mathcal{C}$-feasible* if every function computed by `P` is in $\mathcal{C}$. `P` is *honestly $\mathcal{C}$-feasible* if every subprogram of `P` is $\mathcal{C}$-feasible. `P` is *dishonestly $\mathcal{C}$-feasible*, or *$\mathcal{C}$-dishonest* for short, if `P` is $\mathcal{C}$-feasible, but not honestly $\mathcal{C}$-feasible. Whenever convenient we will leave out the reference to a particular complexity class and talk about feasible programs, honestly feasible programs, etcetera. *End of definition.*

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

$\mathcal{C}$-dishonest programs fall into two groups. One group consists of those programs which only compute functions in $\mathcal{C}$, but the execution of the program require more resources than the complexity class $\mathcal{C}$ admits. The other group consists of programs which always are executed within the resource bounds $\mathcal{C}$ admits, but some subprograms exceed these bounds if executed separately. Typical of the latter group are programs of the form `R; if <test> [Q]` where `R` is an honestly $\mathcal{C}$-feasible program, `<test>` is a test that always fails, and `Q` is an arbitrary program which is not $\mathcal{C}$-feasible. Another example is a program of the form `P;Q` where `Q` runs in time $O(2^x)$, but where `P` is an honestly PF-feasible program which assures that `Q` always is executed on "logarithmically large input". The program `P;Q` is dishonestly PF-feasible.

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 really want to analyse and recognise. But even then, our project is bound to fail.

Definition 2.5. Let $\mathcal{C}$ be a complexity class, let C be a programming language, and let π be a measure on C. The pair (π, C) is called

- $\mathcal{C}$-*sound* if every C-program with π-measure 0 is $\mathcal{C}$-feasible,
- $\mathcal{C}$-*complete* if every honestly feasible C-program has π-measure 0, and
- $\mathcal{C}$-*adequate* if every function in $\mathcal{C}$ can be computed by a C-program with π-measure 0.

End of definition.

Note that if we have a pair (π, C) which is both $\mathcal{C}$-sound and $\mathcal{C}$-adequate, we have a characterisation theorem for the class $\mathcal{C}$, i.e. $f \in \mathcal{C}$ iff f can be computed by a C-program with π-measure 0. Now, let us restrict our discussion to the linear space computable functions for a while, i.e. the Grzegorczyk class $\mathcal{E}^2$. (Recall that $\mathcal{E}^2 =$ LINSPACEF.)

Theorem 2.6. Let C be any programming language extending L_0, and (π, C) be a $\mathcal{E}^2$-sound and $\mathcal{E}^2$-adequate pair. Then (π, C) is $\mathcal{E}^2$-incomplete, that is, there exists an honestly $\mathcal{E}^2$-feasible program $\mathtt{P} \in C$ such that $\pi(\mathtt{P}) > 0$.

Proof. Assume a 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 $\mathcal{E}^2$ satisfying $f_n(x) = 1$ if ρ_n (on the empty input) halts within x steps, and $f_n(x) = 0$ else. It is also well-known that it is undecidable whether ρ_i halts. Since (π, C) is adequate and sound, there is an honestly feasible program $\mathtt{Q} \in C$ with π-measure 0 such that

$$\{\mathtt{Y} = y\}\ \mathtt{Q}\ \{\text{if } \rho_n \text{ does not halt within } y \text{ steps then } \mathtt{Z} = 0 \text{ else } \mathtt{Z} = 1\}$$

Moreover, such a program `Q` can be effectively constructed from n, that is, there exists an algorithm for constructing `Q` from n. Since C extends L_0, the program

```
P :≡ loop X [Q; loop Z [loop V [suc(W)]];
             loop W [suc(V)]]
```

is also in C, where `X`, `V`, `W` are fresh variables. Now, if ρ_n never halts then `loop V [suc(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 `loop V [suc(W)]` and part `loop W [suc(V)]` will be executed each time the body of the outermost loop is executed whenever $\mathtt{Y} = y \geq s$. Each such execution implies that the number stored in the register `V` is at least doubled, and then the function computed into `V` is not in $\mathcal{E}^2$. (For any $f \in \mathcal{E}^2$ we have $f(\vec{x}) \leq q(\vec{x})$ for some polynomial q, but the function computed into `V` cannot be bounded by a polynomial.) 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 (π, C) cannot be complete. For if (π, C) were complete, then ρ_n would never halt if and only if $\pi(\mathtt{P}) = 0$. This would yield an algorithm which decides whether ρ_n halts: Construct `P` from n and then check whether $\nu(\mathtt{P}) > 0$. Such an algorithm does not exist. □

Theorem 2.6 regards loops languages and the linear space computable functions, but similar theorems can be proved for any other interesting programming languages and complexity classes. So, maybe it will be wise to put an end to our project then? In a way we have lost before we even started. Still, we will continue our research project since we belive we can achieve completeness in practice. For a significant amount of natural programs, and in particular for programs that actually will be written for commercial and

industrial purposes, we belive we can find sound and complete measures for interesting complexity classes, that is measures when applied on such natural programs always yield the the best possible answers. Moreover, for sound and adequate pairs (π, C) there might be possible to prove that the measure π is complete for an interesting sublanguage C' of C. This type partial completeness results (or lack of such results) will give a mathematical indication of how well (or bad) we are doing.

3 The μ-measure and some fundamental definitions.

Note 3.1. In the previous section we defined the loop languages L_0, L_1 and L. Any program in any of these languages can be written uniquely in the form $P_1 ; \ldots ; P_k$ such that each P_i is either a loop or an imperative, and where $k = 1$ whenever P is an imperative or a loop. *End of note.*

Definition 3.2. The relations $\prec_{\mathtt{P}}$ and $\xrightarrow{\mathtt{P}}$ are binary relations over $\mathcal{V}(\mathtt{P})$. The relation $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$ holds iff P has a subprogram `loop X [Q]` where `suc(Y)` is a subprogram of Q. The relation $\xrightarrow{\mathtt{P}}$ is the transitive closure of $\prec_{\mathtt{P}}$. We call $\xrightarrow{\mathtt{P}}$ the *control relation* (of P), and if $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$ we say that X controls Y.

α is a clique of degree 1 in the program P iff α is a minimal subset of $\mathcal{V}(\mathtt{P})$ satisfying

(i) there is a 1-principal variable in α, where a variable X is *1-principal* iff $\mathtt{X} \xrightarrow{\mathtt{P}} \mathtt{X}$

(ii) if X is a 1-principal variable in α and $\mathtt{X} \xrightarrow{\mathtt{P}} \mathtt{Y}$, then $\mathtt{Y} \in \alpha$.

Let α be a clique (of any degree) in the program P. The *cover set* $\hat{\alpha} \subset \mathcal{V}(\mathtt{P})$ of the clique α is defined by $\hat{\alpha} = \{\mathtt{X} \mid \mathtt{X} \prec_{\mathtt{P}} \mathtt{Y} \text{ for every } \mathtt{Y} \in \alpha\}$.

α is a clique of degree $n+1$ in the program P iff α is a minimal subset of $\mathcal{V}(\mathtt{P})$ satisfying

(i) there is a $n+1$-principal variable in α, where a variable X is $n+1$-*principal* when there exists a clique β of degree n in P such that $\mathtt{X} \in \hat{\beta}$ and $\mathtt{Y} \xrightarrow{\mathtt{P}} \mathtt{X}$ for some $\mathtt{Y} \in \beta$

(ii) if $\mathtt{X}$ is a $n+1$-principal variable in α and $\mathtt{X} \xrightarrow{\mathtt{P}} \mathtt{Y}$, then $\mathtt{Y} \in \alpha$.

We define the μ-measure $\mu(\mathtt{P})$ of an L_0-*program* $\mathtt{P}$ by

- $\mu(\mathtt{suc(X)}) = 0$ for every variable $\mathtt{X}$.
- Let $\mathtt{P} \equiv \mathtt{Q}_1\mathtt{;Q}_2$. Then $\mu(\mathtt{P}) = \max(\mu(\mathtt{Q}_1), \mu(\mathtt{Q}_2))$.
- Let $\mathtt{P} \equiv \mathtt{loop\ X\ [Q]}$. Then

$$\mu(\mathtt{P}) = \begin{cases} \mu(\mathtt{Q}) + 1 & \text{if } \mathtt{Q} \text{ has a clique of degree } \mu(\mathtt{Q}) + 1, \\ \mu(\mathtt{Q}) & \text{otherwise} \end{cases}$$

End of definition.

We shall elucidate Definition 3.2 by some examples and explanations. Let $\mu(\mathtt{P}) = n > 0$. The functions computed by $\mathtt{P}$ which are not in the Grzegorczyk class $\mathcal{E}^{n+1}$ will be precisely those functions computed into the variables in the n-cliques of $\mathtt{P}$, i.e. if $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\mathtt{Y} = f(\vec{x})\}$ and $\mathtt{Y} \in \alpha$ for some n-clique α of $\mathtt{P}$, then $f \notin \mathcal{E}^{n+1}$ (but $f \in \mathcal{E}^{n+2}$). Let $\mathtt{P}_0$ be the program

```
loop A [suc(B)]; loop B [suc(A)]
```

The μ-measure of $\mathtt{P}_0$ is 0, so every function computed by $\mathtt{P}_0$ is in $\mathcal{E}^2$. There is one 1-clique α in $\mathtt{P}_0$, namely $\alpha = \{\mathtt{A}, \mathtt{B}\}$. The cover set $\hat{\alpha}$ of α is empty.

We put $\mathtt{P}_0$ inside loop controlled by a variable $\mathtt{X}$ and get the program

```
P1 ≡ loop X [ loop A [suc(B)]; loop B [suc(A)] ]
```

Then we have $\mu(\mathtt{P}_1) = 1$ since the body of the outermost loop has μ measure 0 and contains a 1-clique. Still there is only one clique in the program, namely the 1-clique $\alpha = \{\mathtt{A}, \mathtt{B}\}$, but the cover set of α is not empty anymore. Now we have $\hat{\alpha} = \{\mathtt{X}\}$. The functions computed by $\mathtt{P}_1$ into A, B are not in $\mathcal{E}^2$ whereas the function computed into $\mathtt{X}$ is in $\mathcal{E}^2$. We have

$$\{\mathtt{A} = a, \mathtt{B} = b, \mathtt{X} = x\}\ \mathtt{P}_0\ \{\mathtt{A} = f(a,b,x), \mathtt{B} = g(a,b,x), \mathtt{X} = x\}$$

and the reader can check that neither f nor g is bounded by a polynomial, thus neither f nor g is in $\mathcal{E}^2$.

We extend the body of the outermost loop in $\mathtt{P}_1$ and get the program $\mathtt{P}_2 \equiv$

```
loop X [ loop A [suc(C)];
         loop A [suc(B)];
         loop B [suc(A)];
         loop U [suc(V)] ]
```

Still we have only one 1-clique α, but now the variable C is also in the clique, i.e. $\alpha = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$. We still have $\hat{\alpha} = \{\mathtt{X}\}$. Note that the value the program computes into C cannot be bounded by a polynomial in the input since the value the program computes into A is not bounded by such a polynomial. Thus the functions computed into C are not in $\mathcal{E}^2$. The function computed into U and V are in $\mathcal{E}^2$, e.g. we have

$$\{\mathtt{X} = x, \mathtt{U} = u, \mathtt{V} = v\}\ \mathtt{P}_2\ \{\mathtt{V} = v + (x \times u)\}\ .$$

Let us study the program

$$\mathtt{Q}_0 \equiv \texttt{loop Y [} \mathtt{P}_2\texttt{; loop V [suc(U)]] .}$$

The μ measure of $\mathtt{Q}_0$ is also 1, but here we find two cliques of degree 1, the clique $\alpha = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and the clique $\beta = \{\mathtt{U}, \mathtt{V}\}$. Further, we have the cover sets $\hat{\alpha} = \{\mathtt{X}, \mathtt{Y}\}$ and $\hat{\beta} = \{\mathtt{Y}\}$. Now, if we extend the body of the outermost loop of $\mathtt{Q}_0$ such that we get the program

$$\mathtt{Q}_1 \equiv \texttt{loop Y [} \mathtt{P}_2\texttt{; loop V [suc(U)]; loop A [suc(X)]]}$$

then we have a program with μ-measure 2. Why? Well, in the body of the outermost loop of $\mathtt{Q}_1$, i.e. in the program

$$\mathtt{P}_2\texttt{; loop V [suc(U)]; loop A [suc(X)]}$$

we find the 1-clique $\alpha = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ and the cover set $\hat{\alpha} = \{\mathtt{X}\}$. (This clique is inside $\mathtt{P}_2$.) Besides, we see that A controls X since the subprogram

```
loop A [ suc(X) ]
```

is a part of the body. So, a variable (A) in a 1-clique (α) controls a variable (X) in the cover set ($\hat{\alpha}$) of the 1-clique, and hence there is a clique γ of degree 2 in the body. Thus, the program $\mathtt{Q}_1$ has μ measure 2. The sole principal variable of γ is X and every variable X controls, and no other variable, should be in γ, i.e. $\gamma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{X}, \mathtt{V}\}$. The cover set of γ in $\mathtt{Q}_1$ contains only one variable. We have $\hat{\gamma} = \{\mathtt{Y}\}$. The functions the program $\mathtt{Q}_1$ computes into the variables in γ will be hyper exponential, i.e. they will not be bounded by any Kalmár elementary function.

Definition 3.3. Let $\mathcal{V}(\mathtt{P}) = \vec{\mathtt{X}}$. We use $\#_{\mathtt{P}}(\vec{x})$ to denote the number of steps in the execution of the imperative program P on the input $\vec{\mathtt{X}} = \vec{x}$, i.e. the number of primitive imperatives which is executed.

Let π be a measure and let C be a loop language. We say that the pair (π, C) is *sound* (with respect to the Grzegorczyk hierarchy) when

$$\pi(\mathtt{P}) \leq n \;\Rightarrow\; \#_{\mathtt{P}} \in \mathcal{E}^{n+2}$$

for every $n \in \mathbb{N}$ and every $\mathtt{P} \in C$.

We say that the pair (π, C) is *complete* (with respect to the Grzegorczyk hierarchy) when

$$\#_{\mathtt{P}} \in \mathcal{E}^{n+2} \;\Rightarrow\; \pi(\mathtt{P}) \leq n$$

for every $n \in \mathbb{N}$ and every $\mathtt{P} \in C$.

We say that the pair (π, C) is *adequate* (with respect to the Grzegorczyk hierarchy) when every function in $\mathcal{E}^{n+2}$ can be computed by a C-program with π-measure $\leq\ n$ (for every $n \in \mathbb{N}$). *End of definition.*

4 Soundness and completeness of (μ, L_0)

Definition 4.1. We say that a loop `loop X [Q]` with μ-measure $n+1$ is *simple* if Q has μ-measure n. A program $\mathtt{P}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{P}_k$ with μ-measure $n+1$ is *flattened out* if each component $\mathtt{P}_i$ either is a simple loop or has μ-measure strictly less than $n+1$, i.e. $\mu(\mathtt{P}_i) \leq n$. *End of definition.*

For any program P in L_0 we have $\{\mathtt{X} = x\}\mathtt{P}\{\mathtt{X} \geq x\}$. This entails a lot of monotonicity properties for programs in L_0, for instance, if $\mathtt{Q}_0 \ll \mathtt{Q}_1$, then we also have $\mathtt{R}_0\mathtt{;}\ \mathtt{Q}_0\mathtt{;}\ \mathtt{R}_1 \ll \mathtt{R}_0\mathtt{;}\ \mathtt{Q}_1\mathtt{;}\ \mathtt{R}_1$. Such monotonicity properties of L_0 will be used frequently in the sequel, and in particular in the proof of the next lemma.

Lemma 4.2 (Flattening lemma). Let P be an L_0-program such that $\mu(\mathtt{P}) > 0$. Then there exists a flattened out L_0-program $\mathtt{P}'$ such that

- $\mathtt{P}'$ is flattened out
- $\mathtt{P} \ll \mathtt{P}'$
- $\mu(\mathtt{P}') \leq \mu(\mathtt{P})$.

Proof. We define the *rank* of a loop $\mathtt{P}$ denoted $\mathrm{rk}(\mathtt{P})$ by

$$\mathrm{rk}(\mathtt{P}) = \begin{cases} 0 & \text{if } \mu(\mathtt{P}) = 0 \\ k & \text{otherwise} \end{cases}$$

where k is the number of times the word `loop` occurs in $\mathtt{P}$.

We will prove

(Claim) If the loop $\mathtt{P} \equiv \mathtt{loop\ X\ [Q]}$ is not simple, then there are loops $\mathtt{P}_1$ and $\mathtt{P}_2$ such that

(i) $\mathrm{rk}(\mathtt{P}) > \mathrm{rk}(\mathtt{P}_1)$ and $\mathrm{rk}(\mathtt{P}) > \mathrm{rk}(\mathtt{P}_2)$

(ii) $\mu(\mathtt{P}) \geq \mu(\mathtt{P}_1)$ and $\mu(\mathtt{P}) \geq \mu(\mathtt{P}_2)$

(iii) $\mathtt{P} \ll \mathtt{P}_1\,;\, \mathtt{P}_2$

Let us see how the very lemma follows from (Claim). Let $\mathtt{R}_0 \equiv \mathtt{Q}_1\,;\,\ldots\,;\,\mathtt{Q}_m$ be a program with μ-measure $n+1$ which is not flattened out. Then there must be a loop $\mathtt{Q}_i$ among the components $\mathtt{Q}_1\,;\,\ldots\,\mathtt{Q}_m$ such that $\mathtt{Q}_i$ is not a simple loop and $\mu(\mathtt{Q}_i) = n+1$. Clause (iii) of the claim yields loops $\mathtt{P}_1$ and $\mathtt{P}_2$ such that and $\mathtt{P} \ll \mathtt{P}_1\,;\,\mathtt{P}_2$. Let

$$\mathtt{R}_1 \equiv \mathtt{Q}_1\,;\ \ldots\,;\,\mathtt{Q}_{i-1}\,;\,\mathtt{P}_1\,;\,\mathtt{P}_2\,;\,\mathtt{Q}_{i+1}\,;\,\ldots\,;\,\mathtt{Q}_m\,.$$

Then we have $\mathtt{R}_0 \ll \mathtt{R}_1$. If the program $\mathtt{R}_1$ is not flattened out, we can use the same procedure once more to get a program $\mathtt{R}_2$ such that $\mathtt{R}_0 \ll \mathtt{R}_1 \ll \mathtt{R}_2$. And thus we might proceed, constructing programs $\mathtt{R}_1, \mathtt{R}_2, \mathtt{R}_3, \ldots$ such that $\mathtt{R}_0 \ll \mathtt{R}_i$. The process terminates when we encounter a flattened out program $\mathtt{R}_\bullet$. Clause (i) of the claim ensures that we eventually will encounter such a flattened out program. The program $\mathtt{R}_{i+1}$ is the program $\mathtt{R}_i$ where one loop in a sequence of loops (and primitive instructions) is replaced by two loops of strictly lower rank, and thus it follows straightaway from the definition of rank in the beginning of this proof, that the process must terminate. Finally, clause (ii) of the claim ensures that the result of process, i.e. the flattened out program $\mathtt{R}_\bullet$ has μ-measure less or equal to $\mathtt{R}_0$. This proves that the lemma follows from (Claim).

We turn to the proof of (Claim). The proofs splits into two cases. The case (1) where $\mathtt{Q}$ is a loop $\mathtt{loop\ Y\ [R]}$ and the case (2) where $\mathtt{Q}$ is a sequence $\mathtt{Q}_1\,;\ \ldots\,;\,\mathtt{Q}_m$.

Case (1). We have $\mathtt{P} \equiv$ `loop X [loop Y [R]]`. Let `Z` be a fresh variable, let $\mathtt{P}_1 \equiv$ `loop X [loop Y [suc(Z)]`. and let $\mathtt{P}_2 \equiv$ `loop Z [R]`. It is easy to see that (Claim) holds.

Case (2). This is the hard case. We have $\mathtt{P} \equiv$ `loop X [Q]`, where $\mathtt{Q} \equiv \mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_m$ for some $m \geq 2$. Since `P` is not a simple loop, we have $\mu(\mathtt{P}) = \mu(\mathtt{Q}) = n+1$. Thus, there must be a component $\mathtt{Q}_i$ in $\mathtt{Q}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{Q}_m$ such that $\mu(\mathtt{Q}_i) = n+1$. Furthermore, $\mathtt{Q}_i$ has the form $\mathtt{Q}_i \equiv$ `loop Y [R]`, where `R` contains an $n+1$-clique α and $\mathtt{Y} \in \hat{\alpha}$. Now, it must be the case that no `V` in the clique α controls `Y` in the program `Q`, i.e. we have $\mathtt{V} \overset{\mathtt{Q}}{\not\rightarrow} \mathtt{Y}$ for every $\mathtt{V} \in \alpha$. If this were not the case, there would be a clique of degree $n+2$ in `Q`, and then the μ-measure `P` would be $n+2$. Let `Z` be a fresh variable, and let `R'` be `R`, where every loop controlled by a variable in the clique α is deleted. Note that since α has degree $n+1$, the number of occurrences of the word `loop` in `R'` will be strictly less than the number in `R`. (There is no reason to formalise what it means to delete a loop. Any sensible way of deleting a loop will work. The body of the loop may or may not be deleted.) Let

$$\mathtt{P}_1 \ \equiv\ \mathtt{loop\ X\ [\ Q}_0\mathtt{;}\ \ldots\ \mathtt{;\ Q}_{i-1}\mathtt{;\ loop\ Y\ [suc(Z);\ R']\ ;\ Q}_{i+1}\mathtt{;}\ \ldots\ \mathtt{;\ Q}_k\ \mathtt{]}\ .$$

When we execute `P` and $\mathtt{P}_1$ on the same input, the loop `loop Y [suc(Z); R']` in $\mathtt{P}_1$ will be executed exactly as many times as the loop `loop Y [R]` in `P` because the variable `Y` is not controlled by any variable in α. Hence, let

$$\mathtt{P}_2 \ \equiv\ \mathtt{loop\ Z\ [\ Q}_0\mathtt{;}\ \ldots\ \mathtt{;\ Q}_{i-1}\mathtt{;\ R;\ Q}_{i+1}\mathtt{;}\ \ldots\ \mathtt{;\ Q}_k\ \mathtt{]}$$

and we have $\mathtt{P} \ll \mathtt{P}_1\mathtt{;}\ \mathtt{P}_2$. It is easy to see that we also have $\mathrm{rk}(\mathtt{P}_i) < \mathrm{rk}(\mathtt{P})$ and $\mu(\mathtt{P}_i) \leq \mu(\mathtt{P})$ for $i = 1, 2$. This completes the proof of (Claim). □

Theorem 4.3. Every function computed by an L_0-program with μ-measure n is in the Grzegorczyk class $\mathcal{E}^{n+2}$.

Proof. We prove the theorem by induction on n. Assume f is computed by a program `P` with μ-measure 0. Then the program `P` has the form $\mathtt{P}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{P}_k$ (for some $k \geq 1$) where no $\mathtt{P}_i$ contains a clique of degree 1. This entails that the control relation $\overset{\mathtt{P}_i}{\rightarrow}$ is a partial ordering (for $i = 1, \ldots, k$). Use induction on this ordering to prove that any function computed by $\mathtt{P}_i$ is in $\mathcal{E}^2$. It follows that any function computed by `P` is in $\mathcal{E}^2$, since $\mathcal{E}^2$ is closed under composition. (The induction over the partial ordering $\overset{\mathtt{P}_i}{\rightarrow}$ is tedious and

contains some unpleasant details, but no big surprises show up and everything is quite straightforward. The reader interested in the details should consult Kristiansen-Niggl [13] where such an induction is used to prove a similar result.)

We turn to the induction step. Assume f is computed by a program $\mathtt{P}$ with μ-measure $n+1$. By Lemma 4.2 there exists a flattened out program $\mathtt{P}'$ such that $\mathtt{P} \ll \mathtt{P}'$ and $\mu(\mathtt{P}') \leq n+1$. The program $\mathtt{P}'$ has the form $\mathtt{P}_1\mathtt{;}\ \ldots\ \mathtt{;}\ \mathtt{P}_k$ (for some $k \geq 1$) where $\mathtt{P}_i$ is a simple loop or has μ-measure strictly less than $n+1$ (for $i = 1, \ldots, k$). It suffices to prove that each $\mathtt{P}_i$ only computes functions in $\mathcal{E}^{n+3}$. If $\mathtt{P}_i$ has μ-measure strictly less than $n+1$, it follows straightaway from the inductions hypothesis that $\mathtt{P}_i$ does not compute functions outside $\mathcal{E}^{n+3}$. If $\mathtt{P}_i$ is a simple loop, then $\mathtt{P}_i$ has the form `loop X [Q]` where $\mu(\mathtt{Q}) = n$. By the induction hypothesis every function computed by $\mathtt{Q}$ is in $\mathcal{E}^{n+2}$. Any function computed by $\mathtt{P}_i$ can be defined by one application of simultaneous recursion over functions computed by $\mathtt{Q}$. Hence any function computed by $\mathtt{P}_i$ is in $\mathcal{E}^{n+3}$ since one application of simultaneous recursion over functions in $\mathcal{E}^{n+2}$ yields a function in $\mathcal{E}^{n+3}$. □

Definition 4.4. We define the sequence of functions $K_0, K_1, K_2, \ldots$ from by $K_0(x) = 2(x+1)$ and $K_{n+1}(x) = K_n^x(0)$.

We say that a Hoare statement $\{A\}\mathtt{P}\{B\}$ holds *almost everywhere* if there exists a fixed number k such that $\{A\}\mathtt{P}\{B\}$ holds whenever all the inputs to the program $\mathtt{P}$ is greater or equal to k. We put the tag (a.e) next to a Hoare statement to mark that the statement just is supposed to hold almost everywhere. (A Hoare statement without such a tag is of course supposed to hold for all possible inputs.) *End of definition.*

Example 4.5. Let $\mathtt{P}$ be the program `loop Z [loop X [suc(Y)]]`. We have $\{\mathtt{X} = x\}\mathtt{P}\{\mathtt{Y} \geq x\}$ (a.e) because the statement holds when all inputs is greater or equal to 1. The statement may not be true for inputs where $\mathtt{Z} = 0$.So the statement does not hold for all possible inputs, but it does hold almost everywhere. *End of example.*

Lemma 4.6. We have $K_{n+1} \notin \mathcal{E}^{n+2}$ for $n \in \mathbb{N}$ (but we do have $K_n \in \mathcal{E}^{n+2}$).

Proof. This proof is a standard exercise. We skip the details. □

Lemma 4.7. Let $\mathtt{P}$ be a loop, and let α be a clique of degree n in $\mathtt{P}$. For every $\mathtt{Y} \in \hat{\alpha}$ and $\mathtt{X} \in \alpha$ we have

$$\{\mathtt{Y} = y\}\ \mathtt{P}\ \{\mathtt{X} \geq K_n(y \dot{-} 1)\}\ \text{(a.e)}$$

Proof. We prove this lemma by induction on n. Assume $n = 1$. (There is no such thing as a 0-clique.) Let α be an arbitrary 1-clique in $\mathtt{P}$ and let $\mathtt{A}$ be a principal variable of α and let $\mathtt{B} \in \alpha$ be such that $\mathtt{A} \xrightarrow{\mathtt{P}} \mathtt{B}$ and $\mathtt{B} \xrightarrow{\mathtt{P}} \mathtt{A}$. (Such $\mathtt{A}$ and $\mathtt{B}$ exist when α is a 1-clique.) Let

$$\mathtt{Q} \equiv \mathtt{loop\ A\ [suc(B)];\ loop\ B\ [suc(A)]}\ .$$

Then

$$\{\mathtt{A} = a, \mathtt{B} = b\}\ \mathtt{Q}\ \{A = a + b + a \overset{\text{(a.e)}}{\geq} 2(a+1) = K_0(a)\}\ .$$

Further, let $\mathtt{Q}_1 \equiv \mathtt{loop\ Y\ [Q]}$. Then we have

$$\{\mathtt{Y} = y, \mathtt{A} = a\}\ \mathtt{Q}_1\ \{A \geq K_0^y(a) \geq K_0^y(0) = K_1(y)\} \quad \text{(a.e)}$$

and if $\mathtt{Y}$ is any variable in the cover set $\hat{\alpha}$ and $\{\mathtt{Y} = y\}\mathtt{P}\{\mathtt{A} = a\}$ then $a \geq K_1(y)$. Hence we have $\{\mathtt{Y} = y\}\mathtt{P}\{\mathtt{A} \geq K_1(y)\}$(a.e) for any $\mathtt{Y} \in \hat{\alpha}$. Now, $\mathtt{A}$ is a principal variable of α, so if $\mathtt{X} \in \alpha$ then $\mathtt{A} \xrightarrow{\mathtt{P}} \mathtt{X}$. It follows that we have $\{\mathtt{Y} = y\}\mathtt{P}\{\mathtt{X} \geq K_1(y \dot{-} 1)\}$ (a.e) for any $\mathtt{X} \in \alpha$ and $\mathtt{Y} \in \hat{\alpha}$. This concludes the proof for $n = 1$, and we can turn to the induction step.

Let α be any clique of degree $n + 1$ in the loop $\mathtt{P}$. Then there exists a subprogram $\mathtt{P}_0 \equiv \mathtt{loop\ U\ [Q]}$ such that $\mu(\mathtt{Q}) = n$, $\mu(\mathtt{P}_0) = n + 1$ and an $n+1$-clique α_0 is in $\mathtt{Q}$. The clique α_0 is a subclique of α, i.e. $\alpha_0 \subset \alpha$. Now, $\mathtt{Q}$ has the form $\mathtt{Q}_1\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{Q}_m$ for some $m \geq 2$. Pick a component $\mathtt{Q}_i$ among $\mathtt{Q}_1\mathtt{;}\ \ldots\mathtt{;}\ \mathtt{Q}_m$ such that $\mu(\mathtt{Q}_i) = n$, and there is an n-clique β in $\mathtt{Q}_i$ such that $\mathtt{A} \xrightarrow{\mathtt{Q}} \mathtt{B}$ for some $\mathtt{A} \in \beta$ and $\mathtt{B} \in \hat{\beta}$. Such a $\mathtt{Q}_i$ exists since $\mu(\mathtt{P}_0) = n + 1$. Furthermore, $\mathtt{B}$ is a principal variable for the clique α_0. By the induction hypothesis we have

$$\{\mathtt{B} = b\}\mathtt{Q}_i\{\mathtt{A} \geq K_n(b \dot{-} 1)\}\ .\ \text{(a.e)}$$

Let $\mathtt{Y}$ be any variable in the cover set $\hat{\alpha}$ and let $\mathtt{P}_1 \equiv \mathtt{loop\ Y\ [B:=\ A;\ Q}_i\mathtt{]}$. ($\mathtt{B:=\ A}$ is ordinary assignment. It does not affect our argument that assignment is not a part of the language L_0.) First, we note that

$$\{\mathtt{Y} = y, \mathtt{A} = a\}\mathtt{P}_1\{A \leq \overline{K}_n^y(a)\} \quad \text{(a.e)}$$

where $\overline{K}_n(x) = K_n(x \dot{-} 1)$. Further, we note that $K_n(K_n(x) \dot{-} 1) \geq K_n(K_n(x \dot{-} 1))$. Hence

$$\overline{K}_n^y(a) \ \geq\ K_n^y(a \dot{-} y) \ \geq\ K_n^y(0) \ =\ K_{n+1}(y)\ .$$

Altogether this entails that $\{\mathtt{Y} = y\}\mathtt{P}_1\{\mathtt{A} \geq K_{n+1}(y)\}$ (a.e). It is easy to see that $\mathtt{P}_1 \ll \mathtt{P}$, and thus we have proved that $\{\mathtt{Y} = y\}\mathtt{P}\{\mathtt{A} \geq K_{n+1}(y)\}$ (a.e) holds for any $\mathtt{Y} \in \hat{\alpha}$ and the variable $\mathtt{A} \in \alpha$. Now, $\mathtt{A} \xrightarrow{\mathtt{P}} \mathtt{B}$ and $\mathtt{B}$ is a principal variable of the clique α_0. The variable $\mathtt{B}$ will control (in $\mathtt{P}$) any variable in the clique α, and thus $\mathtt{A}$ controls (in $\mathtt{P}$) any variable in the clique α. Moreover, $\mathtt{P}$ is a loop. It follows that $\{\mathtt{Y} = y\}\mathtt{P}\{\mathtt{X} \geq K_{n+1}(y \dot{-} 1)\}$ (a.e) for any $\mathtt{X} \in \alpha, \mathtt{Y} \in \hat{\alpha}$. □

Theorem 4.8 (Soundness and Completeness of (μ, L_0)). We have

$$\mu(\mathtt{P}) \leq n \quad \Leftrightarrow \quad \#_{\mathtt{P}} \in \mathcal{E}^{n+2}$$

for every $n \in \mathbb{N}$ and every $\mathtt{P} \in L_0$.

Proof. Suppose $\mu(\mathtt{P}) \leq n$. Let $\mathtt{A}$ be a fresh variable, and let $\mathtt{Q}$ be $\mathtt{P}$ where each occurrence of an imperative on the form `suc(X)` is replaced by the program `suc(X); suc(A)`. Let f be the function which $\mathtt{Q}$ computes into $\mathtt{A}$, i.e. we have

$$\{\vec{\mathtt{X}} = \vec{x}, \mathtt{A} = a\}\ \mathtt{Q}\ \{\mathtt{A} = f(\vec{x}, a)\}\ .$$

It is easy to see that $\mathtt{Q}$ has μ-measure less or equal n, and thus we have $f \in \mathcal{E}^{n+2}$ by Theorem 4.3. Now, $f(\vec{x}, a) = a + \#_{\mathtt{P}}(\vec{x})$. It follows that $\#_{\mathtt{P}} \in \mathcal{E}^{n+2}$

Suppose $\mu(\mathtt{P}) > n$. Then a subprogram $\mathtt{Q}$ of $\mathtt{P}$ is a loop containing a clique of degree $n + 1$. By Lemma 4.7 there will be variables $\mathtt{Y}$ and $\mathtt{X}$ such that

$$\{\mathtt{Y} = y\}\ \mathtt{Q}\ \{\mathtt{X} \geq K_{n+1}(y \dot{-} 1)\}\ \text{(a.e)}\ .$$

The function K_{n+1} is not in $\mathcal{E}^{n+2}$. (Lemma 4.6.) It follows that $\#_{\mathtt{P}}$ cannot be in $\mathcal{E}^{n+2}$, i.e. we have $\#_{\mathtt{P}} \notin \mathcal{E}^{n+2}$ □

The μ-measure introduced above is considerably more sophisticated than a similar measure (also called μ-measure) in Kristiansen-Niggl [13]. The measure in [13] is sound, but not complete, for a language corresponding to L_0.

5 The ν-measure and how to deal with assignment

Recall that L_1 is the language L_0 extended by the assignment imperative `..  := ...` An interesting feature of the theory we are developing is that the measures on programs still will be well defined when we extend the language in a natural way. The μ-measure is still well defined for L_1, but the next example shows that the pair (μ, L_1) is not sound with respect to the Grzegorczyk hierarchy.

Example 5.1. The program

```
loop X [ loop Y [suc(Z)]; Y:= Z ]
```

computes a function into `Y` which are not bounded by a polynomial, and thus not bounded by any function in $\mathcal{E}^2$. (So the number of steps in an execution of the program is not bounded by a function in $\mathcal{E}^2$ either.) Still the μ-measure of the program is 0. *End of example.*

So we need a measure ν that makes the pair (ν, L_1) sound. We will achieve ν by redefining the control relation $\overset{\mathtt{P}}{\rightarrow}$. In any other respect the ν-measure will be defined exactly as the μ-measure, e.g. the notions of clique, cover set etc. will not be redefined.

The naive way to redefine the control relation would be to say that the relation $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$ holds if `P` has a subprogram `loop X [Q]` where `suc(Y)` is a subprogram of `Q` or if `Y:= X` is a subprogram of `P`; and then define the relation $\overset{\mathtt{P}}{\rightarrow}$ as the transitive closure of $\prec_{\mathtt{P}}$. If we do so, the pair (ν, L_1) will certainly be sound, still this solution turns out to be thoroughly unsatisfactory. Example 5.2 shows a natural and innocent little program of very low computational complexity, which receives ν-measure 1 if we assume this naive definition of the control relation. The example speaks for itself, moreover, the naive definition would not enable us to prove several of the theorems in the sequel.

Example 5.2. Let `P` be the program

```
loop X [ C:= A; A:= B; B:= C ]
```

Then we have $\{\mathtt{X} = x, \mathtt{A} = a, \mathtt{B} = b\}\mathtt{P}\{\mathtt{A} = f(x, a, b,)\}$ where $f(x, a, b)$ equals a if x is even and b if x is odd. *End of example.*

So, how should one define define the control relation ? The next definition shows a way which turns out to be satisfactory.

Definition 5.3. The relations $\prec_{\mathtt{P}}$ and $\xrightarrow{\mathtt{P}}$ are binary relations over $\mathcal{V}(\mathtt{P})$. The relation $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$ holds iff P has a subprogram `loop X [Q]` where `suc(Y)` is a subprogram of Q. Let $\mathtt{X:=}_{\mathtt{P}}\mathtt{Y}$ denote that `X:= Y` is a subprogram of P. The relation $\xrightarrow{\mathtt{P}}$ is the smallest relation such that

- if $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$, then $\mathtt{X} \xrightarrow{\mathtt{P}} \mathtt{Y}$
- $\xrightarrow{\mathtt{P}}$ is transitive
- if $\mathtt{X:=}_{\mathtt{P}}\mathtt{Y}$ and $\mathtt{Z} \xrightarrow{\mathtt{P}} \mathtt{Y}$, then $\mathtt{Z} \xrightarrow{\mathtt{P}} \mathtt{X}$

We keep the definitions from 3.2 of principal variable, n-clique and cover set (in P). These notions depends solely on the relations $\prec_{\mathtt{P}}$ and $\xrightarrow{\mathtt{P}}$. Finally, we define the ν-measure as we defined the μ-measure, i.e. we define the ν-measure of a program P, denoted $\nu(\mathtt{P})$, by

- $\nu(\mathtt{imp}) = 0$ for every imperative `imp`.
- Let $\mathtt{P} \equiv \mathtt{Q}_1\mathtt{;}\ \mathtt{Q}_2$. Then $\nu(\mathtt{P}) = \max(\nu(\mathtt{Q}_1), \nu(\mathtt{Q}_2))$.
- Let $\mathtt{P} \equiv \mathtt{loop\ X\ [Q]}$. Then

$$\nu(\mathtt{P}) = \begin{cases} \nu(\mathtt{Q}) + 1 & \text{if Q has a clique of degree } \nu(\mathtt{Q}) + 1 \\ \nu(\mathtt{Q}) & \text{else} \end{cases}$$

End of definition.

A bit informally stated, the ν-measure is the μ-measure with the extension: if `X:= Y` occur in P, then X inherits all the $\xrightarrow{\mathtt{P}}$-predecessors of Y. The program in Example 5.1 has ν-measure 1, whereas the program in Example 5.2 has ν-measure 0. Everything seems fine, and as we soon will prove, the pair (ν, L_1) is indeed sound.

Programs in L_1 do not possess the same nice monotonicity properties as the the programs in L_0. E.g. we do not have $\{\mathtt{X} = x\}\mathtt{P}\{\mathtt{X} \geq x\}$ for arbitrary $\mathtt{P} \in L_1$ and $\mathtt{X} \in \mathcal{V}(\mathtt{P})$. The program in Example 5.2 yields a counterexample.

This clutters the proof of soundness for the pair (ν, L_1). We will introduce a new loop language L_M with convenient monotonicity properties, redefine the measure ν for this language, and then prove that the pair (ν, L_M) is sound with respect to the Grzegorczyk hierarchy. It will follow easily that the pair (ν, L_1) is also sound.

Definition 5.4. Let L_M be the loop language with the imperatives `suc(X)` and `X:= max(X,Y)` for any variables `X` and `Y`. The semantics of the imperative `X:= max(X,Y)` is indicated by the syntax, i.e. we have $\{\mathtt{X} = x, \mathtt{Y} = y\}\mathtt{X:= max(X,Y)}\{\mathtt{X} = \max(x, y), \mathtt{Y} = y\}$. Note that `X:= max(Y,Z)` is not an imperative of L_M.

We define the measure ν for L_M exactly as we defined ν for L_1, except that $\mathtt{X:=_P Y}$ now denotes that `X:= max(X,Y)` is a subprogram of `P`. (There is no need to repeat the definition.) *End of definition.*

Lemma 5.5 (Flattening lemma). Let `P` be an L_M-program such that $\mu(\mathtt{P}) > 0$. Then there exists a flattened out L_M-program $\mathtt{P}'$ such that

- $\mathtt{P}'$ is flattened out
- $\mathtt{P} \ll \mathtt{P}'$
- $\nu(\mathtt{P}') \leq \nu(\mathtt{P})$.

Proof. This lemma is similar to Lemma 4.2. The proofs of the two lemmas are also very similar. First we define "rank" as we did in the proof of 4.2, i.e. the *rank* of a loop `P` denoted $\mathrm{rk}(\mathtt{P})$ by

$$\mathrm{rk}(\mathtt{P}) = \begin{cases} 0 & \text{if } \nu(\mathtt{P}) = 0 \\ k & \text{otherwise} \end{cases}$$

where k be the number of times the word `loop` occurs in `P`. Then we need to prove a claim similar to the claim in proof of 4.2.

(Claim) If the loop $\mathtt{P} \equiv$ `loop X [Q]` is not simple, then there exist loops $\mathtt{P}_1$ and $\mathtt{P}_2$ such that

(i) $\mathrm{rk}(\mathtt{P}) > \mathrm{rk}(\mathtt{P}_1)$ and $\mathrm{rk}(\mathtt{P}) > \mathrm{rk}(\mathtt{P}_2)$

(ii) $\nu(\mathtt{P}) \geq \nu(\mathtt{P}_1)$ and $\nu(\mathtt{P}) \geq \nu(\mathtt{P}_2)$

(iii) $\mathtt{P} \ll \mathtt{P}_1\,;\, \mathtt{P}_2$

The desired result follows from (Claim) by the same argument we used above. The proof (Claim) is also very similar to the corresponding proof above. The proofs splits into two cases. Case (i) is more or less identical to the corresponding case above. Case (ii) is also similar to the corresponding case above, but an additional argument is required to establish $\mathrm{rk}(\mathtt{P}_1) < \mathrm{rk}(\mathtt{P})$. The augment is quite easy: When a program contains a clique (of any degree) there will be a least one loop in the program. (This is obvious, a program which is just a sequence of imperatives cannot have a clique.) The program $\mathtt{P}_1$ is the program

$$\texttt{loop X [Q}_0\texttt{; ... ; Q}_{i-1}\texttt{; loop Y [suc(Z); R']; Q}_{i+1}\texttt{; ... ; Q}_k\texttt{]}$$

where the subprogram `R'` is obtained by removing loops form a certain program `R`. All loops controlled by the variables in a certain clique in `R` are removed, so there will be at least one loop to remove. It follows that $\mathrm{rk}(\mathtt{P}_1) < \mathrm{rk}(\mathtt{P})$. This concludes the proof of Lemma 5.5. □

Theorem 5.6 (Soundness of (ν, L_1)). We have

$$\mu(\mathtt{P}) \leq n \quad \Rightarrow \quad \#_{\mathtt{P}} \in \mathcal{E}^{n+2}$$

for every $n \in \mathbb{N}$ and every $\mathtt{P} \in L_1$.

Proof. Let `P` be a L_1 program with ν-measure n. Obviously there exists a L_M program $\mathtt{P}'$ with ν-measure n such that $\mathtt{P} \ll \mathtt{P}'$. By Lemma 5.5 we can assume that $\mathtt{P}'$ is flattened out. Henceforth we can proceed as in the proof of Theorem 4.3, and prove by induction on n that every function computed by an L_1-program with ν-measure n is in the Grzegorczyk class $\mathcal{E}^{n+2}$. Thereafter we can proceed as the left-right direction in the proof of Theorem 4.8. □

Unfortunately, the pair (ν, L_1) is not complete. The program

$$\mathtt{P} \equiv \texttt{loop X [loop A [suc(B)]]}$$

has ν-measure 0, and thus $\#_{\mathtt{P}} \in \mathcal{E}^2$ by Theorem 5.6. We expand `P` to the program `Q` $\equiv$

```
loop X
[ loop A [suc(B)]
  C:= A; A:= B; A:= C; (* swap the values of A and B *)
  C:= A; A:= B; A:= C; (* swap the values of A and B *) ]
```

and get a program with ν-measure 1, but obviously we also have $\#_{\mathtt{Q}} \in \mathcal{E}^2$. Thus, the pair (ν, L_1) is not complete. On the other hand, the language L_1 is not adequate, and hence we cannot use the argument in the proof of Theorem 2.6 to prove that no complete measure for L_1 exist. We have an open problem.

Open problem 5.7. Does there exist a measure π such that the pair (π, L_1) is sound and complete with respect to the Grzegorczyk hierarchy?

6 Soundness and adequacy of (ν, L)

Recall that L is the loop language where `suc(..)`, `pred(..)`, `nil(..)` and `..:= ..` are the imperatives. Note that the ν-measure is well defined for L-programs.

Theorem 6.1 (Soundness of (ν, L)). We have

$$\nu(\mathtt{P}) \leq n \quad \Rightarrow \quad \#_{\mathtt{P}} \in \mathcal{E}^{n+2}.$$

for every $n \in \mathbb{N}$ and every $\mathtt{P} \in L$.

Proof. Assume $\mathtt{P} \in L$ and $\nu(\mathtt{P}) \leq n$. Let `Z` be a fresh variable and let $\mathtt{P}_0$ be `P` where each occurrence of a an imperative `imp` is replaced by `imp; suc(Z)`. Let $\mathtt{P}_1 \equiv$ `nil(Z);` $\mathtt{P}_0$. It is easy to see that $\nu(\mathtt{P}_1) = \nu(\mathtt{P})$ and that $\mathtt{P}_1$ computes the function $\#_{\mathtt{P}}$. It is also easy to see that there exists a L_M-program $\mathtt{P}_2$ such that $\mathtt{P}_1 \ll \mathtt{P}_2$ and $\nu(\mathtt{P}_2) = \nu(\mathtt{P}_1)$. (Delete each string on the form `nil(X);` and each string on the form `pred(X);` from $\mathtt{P}_1$. Let `Q` denote the resulting program. Let $\mathtt{P}_2$ be `Q` where each imperative on the form `X:= Y` is replace by `X:= max(X,Y)`.) By lemma 5.5 we can assume that $\mathtt{P}_2$ is flattened out. Henceforth we can proceed as in the proof of Theorem 4.3, and use induction on n to prove that every function computed by $\mathtt{P}_2$ is in $\mathcal{E}^{n+2}$. It follows that $\#_{\mathtt{P}} \in \mathcal{E}^{n+2}$. □

Theorem 6.2 (Adequacy of (ν, L)). Every function in $\mathcal{E}^{n+2}$ can be computed by a L-program with ν-measure n.

Proof. We skip this proof. A proof of a similar result can be found in Kristiansen-Niggl [13]. □

It follows straightaway from Theorem 6.1 and Theorem 6.2 that we can characterise the Grzegorczyk hierarchy at and above the linear space level in terms of L-programs. We have $\mathcal{E}^{n+2} = \mathcal{L}^n$ for all $n \in \mathbb{N}$ where $\mathcal{L}^n$ is the set of function computed by an L-program with ν-measure n. It also follows from the two theorems (and Theorem 2.6) that the pair (ν, L) cannot be complete with respect to the Grzegorczyk hierarchy.

There are a number of rather obvious, but still interesting, ways to extend the language L such that the pair (ν, L) remains sound (and of course adequate). We can add if-then and if-then-else structures:

```
..... if <test> then [......] else [.....] .......
```

The expressions permitted in `<test>` can be very sophisticated. The only requirement is that `<test>` must be a relation in $\mathcal{E}^2$ and that the evaluation of `<test>` does not mess the content of any registers. None of these requirements are very restrictive. We can add a statements which will terminate loops, e.g. the construction

```
..... loop X [ .......; exit <test>; .........] .....
```

where the loop governed by `X` terminates immediately when `<test>` is evaluated to true. Further, can we extend our arsenal of imperatives with a number of operations usually available in Pascal-like languages, e.g.

- $\{\mathtt{X} = x\}$ `sqrt(X)` $\{\mathtt{X} = \lfloor\sqrt{x}\rfloor\}$ (square root)
- $\{\mathtt{X} = x, \mathtt{Y} = y\}$ `div(X,Y)` $\{\mathtt{X} =$ if $y > 0$ then $\lfloor\frac{x}{y}\rfloor$ else $0, \mathtt{Y} = y\}$ (division)
- $\{\mathtt{X} = x, \mathtt{Y} = y\}$ `mod(X,Y)` $\{\mathtt{X} = x - (\lfloor\frac{x}{y}\rfloor \times y), \mathtt{Y} = y\}$ (remainder)
- $\{\mathtt{X} = x, \mathtt{Y} = y\}$ `sub(X,Y)` $\{\mathtt{X} = x \dot{-} y, \mathtt{Y} = y\ \}$ (modified subtraction)

Note that all the proposed imperatives fulfil the requirements: (1) the imperative will never increase the value of a register, and (2) the imperative does not compute functions outside $\mathcal{E}^2$. Indeed we are free to extend the language L by any imperative not violating these two requirements, and still retain soundness of the pair (ν, L). We can also permit constants anywhere it makes sense, e.g. as the second parameter in the subtraction imperative or

at the right hand side of an assignment. The imperative `X:=` c, where c is a fixed natural number, should not be viewed as an assignment, but as a basic imperative setting the value of `X` to c (like `nil(X)` sets the value to 0). Thus, introduction of imperatives of the form `X:=` c, where c is a constant, will not affect the definition of the ν-measure. Perhaps it is not entirely obvious that the pair (ν, L) still will be sound when such assignments are permitted, but it will.

No doubt, many other extensions which will bring L closer to a high level Pascal-like language and still not violate soundness of the pair (ν, L) are possible. The following properties of L are used in the proof of Theorem 6.1: (1) For very program $\mathtt{P} \in L$ there exists a program $\mathtt{P}' \in L_M$ such that $\mathtt{P} \ll \mathtt{P}'$. (2) No imperative in L computes functions outside $\mathcal{E}^2$. Roughly speaking, these two properties of L (together with a number of other *obvious* properties) are all we need to prove soundness of the pair (ν, L).

When we include the few trivial extensions we have proposed above in L, then L becomes a nice little language which engineers and students of applied computer science will recognise. Of course the language is not well equipped in many respects, but the language is actually useful as a tool for analysing the complexity of number-theoretic problems. Let us take the decision problem PRIME as an example, i.e. is it the case whether x is prime. A natural algorithm that solves the problem is easy to implement in L. The first algorithm which falls into our mind is perhaps to check for each number u from 2 to $\lfloor\sqrt{(x)}\rfloor$ whether u divides x. As soon as we find a number in the given interval which divides x, we know that x is not prime. If we can search through the whole interval without finding such a number, we know that x is prime. It is straight forward to implement this algorithm in the extended version of L. Let $\mathtt{P} \equiv$

```
Y:= X; sqrt(Y); sub(Y,1); U:= 2;
loop Y [ Z:= X; mod(Z,U); exit Z=0; Z:= 1; suc(U) ]
```

It is easy to check that implementation is correct, i.e. that we have

$$\{\mathtt{X} = x > 1\}\ \mathtt{P}\ \{\mathtt{Z} = \text{ if } x \textit{ is prime} \text{ then 1 else 0}\}\ .$$

Then we can check the ν-measure of `P`. This is a mechanical process, and in the present case it is easy to see that $\nu(\mathtt{P}) = 0$. Hence, we have established that PRIME is a problem in LINSPACE by a straightforward implementation of

the first algorithm which fell into our minds (accompanied by some trivial and mechanisable argumentation). The standard way to establish that a problem is in LINSPACE is to find a suitable algorithm, implement the algorithm on a Turing machine, and then verify that the Turing machine work in linear time in the length of the input. Now, this is not easy unless you are a trained complexity theorist.

So, a language like L together with a measure like μ is a useful tool when it comes to analysing the complexity of natural problems. Other formalisms developed in implicit computational complexity theory do not provide such tools. (Well, at least not so useful tools.) We will probably be better off analysing the computational complexity of a problem by ordinary mathematical reasoning than to trying to find a program which solves the problem in a formalism of e.g. Bellantoni-Cook.

7 Stack programs computing on the symbols of an alphabet

To capture the important complexity classes P and PF we need the stack programming language introduced in Kristiansen-Niggl [13]. We presuppose an arbitrary but fixed alphabet Σ and define a programming language computing on stacks over Σ. Intuitively, variables serve as *stacks*, each holding an arbitrary word over Σ. Like loop languages, stack languages are based on loops where the (maximal) number of times the body of a loop will be executed is determined before the execution of a loop starts.

Definition 7.1 *(Stack programs).* We have an alphabet Σ. We have an infinite supply of program variables (stacks). We will normally use $\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{U}, \mathtt{V}$ with or without subscript and superscripts to denote variables. For any variables $\mathtt{X}$, $\mathtt{Y}$ and any $\mathtt{a} \in \Sigma$ we have the following *imperatives* (primitive instructions): `push(a,X)`, `pop(X)`, `nil(X)`, `X:= Y`.

A *stack language* is a set of programs generated by *some* of the following syntactical rules:

(I) Every *imperative* is a stack program.

(II) If $\mathtt{P}_1, \mathtt{P}_2$ are stack programs, then so is the *sequence* statement $\mathtt{P}_1\,\mathtt{;}\,\mathtt{P}_2$.

(III) If P is a stack program, then so is every *conditional* statement

$$\texttt{if top(X)} \equiv \texttt{a [P]}$$

.

(IV) If P is a stack program with no occurrence of the variable X in an imperative, then so is the *loop* statement `foreach X [P]`.

The semantics of stack programs are as follows:

- the subprograms in a sequence $\texttt{P}_1\texttt{;} \ldots \texttt{;}\ \texttt{P}_k$ are executed one by one from the left to the right (Note that 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, an imperative, or a conditional.)
- the semantics of imperatives and conditionals are as follows:
  - `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]` has a "call by value" semantics: let `U` be a fresh variable and let `P′` be `P` where each occurrence of `X` is replaced by `U`, then the semantics of `foreach X [P]` is given by the program

$$\texttt{U:= X;}\ \underbrace{\texttt{P'; pop(U); } \ldots \texttt{; P'; pop(U)}}_{|\texttt{X}| \text{ times}}$$

S_0 is the stack language generated by rule (I), (II) and (IV) from the sole imperative `push(a,X)` (for any stack `X` and any `a` $\in \Sigma$). (So, there will be no conditionals in S_0-programs.) S_1 is the stack language generated by rule (I), (II), (III) and (IV) from the imperatives `push(a,X)` and `X:= Y` (for any variables `X`, `Y` and any `a` $\in \Sigma$). S is the full stack language containing all the imperatives listed above (`push(a,X)`, `pop(X)`, `nil(X)`, `X:= Y`), and closed under rule (I), (II), (III) and (IV). *End of definition.*

So, when a loop `foreach X [P]` is executed, the subprogram `P` works on a local copy of the stack `X`, and when the loop terminates the stack `X` has the same value as immediately before the execution of the loop started. This call-by-value semantics ensures that S_0-programs are *length-monotonic*, that is, if $\boldsymbol{P}$ is a S_0-program with variables $\vec{\boldsymbol{X}}$, then $\{\vec{\boldsymbol{X}}{=}\vec{w}\}\ \boldsymbol{P}\ \{\vec{\boldsymbol{X}}{=}\vec{u}\}$ implies $|\vec{w}| \leq |\vec{u}|$ (component-wise), and if $|\vec{w}| \leq |\vec{w}'|$, $\{\vec{\boldsymbol{X}}{=}\vec{w}\}\ \boldsymbol{P}\ \{\vec{\boldsymbol{X}}{=}\vec{u}\}$ and $\{\vec{\boldsymbol{X}}{=}\vec{w}'\}\ \boldsymbol{P}\ \{\vec{\boldsymbol{X}}{=}\vec{u}'\}$, then $|\vec{u}| \leq |\vec{u}'|$. These monotonicity properties are essential in the proofs of the theorems in the sequel.

We will now proceed and define the μ-measure and the ν-measure for stack programs. The definitions are identical to the corresponding definitions for loop programs, except that the `push`-imperative has taken the place of the `suc`-imperative, and that the loop `foreach ...` has taken the place of the loop `loop ...`.

Definition 7.2. The relations $\prec_{\mathtt{P}}$ and $\xrightarrow{\mathtt{P}}$ are binary relations over $\mathcal{V}(\mathtt{P})$. The relation $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$ holds iff

> `P` has a subprogram `foreach X [Q]` where `Q` has a subprogram on the form `push(b,Y)`.

The relation $\xrightarrow{\mathtt{P}}$ is the transitive closure of $\prec_{\mathtt{P}}$.

Cliques, cover sets, etcetera are defined exactly as for loop programs (definition 3.2). We define the the μ-measure $\mu(\mathtt{P})$ of an S_0-program `P` by

- $\mu(\mathtt{push(a,X)}) = 0$ for every variable `X` and every $\mathtt{a} \in \Sigma$.
- Let $\mathtt{P} \equiv \mathtt{Q}_1\mathtt{;}\mathtt{Q}_2$. Then $\mu(\mathtt{P}) = \max(\mu(\mathtt{Q}_1), \mu(\mathtt{Q}_2))$.
- Let $\mathtt{P} \equiv \mathtt{foreach\ X\ [Q]}$. Then

$$\mu(\mathtt{P}) = \begin{cases} \mu(\mathtt{Q}) + 1 & \text{if } \mathtt{Q} \text{ has a clique of degree } \mu(\mathtt{Q}) + 1 \\ \mu(\mathtt{Q}) & \text{else} \end{cases}$$

End of definition.

We can now state and prove theorems, which in a perfectly good sense are soundness theorems, or completeness theorems, for stack languages and their measures.

Theorem 7.3 (Soundness and Completeness of (μ, S_0)). We have

$$\mu(\mathtt{P}) \leq n \quad \Leftrightarrow \quad \#_{\mathtt{P}}(\vec{x}) \leq f(|\vec{x}|) \text{ for some } f \in \mathcal{E}^{n+2}.$$

for every $n \in \mathbb{N}$ and every $\mathtt{P} \in S_0$.

Proof. Analogous to the proof of 4.8. □

Corollary 7.4. Let $\mathtt{P} \in S_0$. Every function computed by P is in PF if and only if P has μ-measure 0.

Proof. Assume P has μ-measure 0. By Theorem 7.3 there is a function $f \in \mathcal{E}^2$ such that $\#_{\mathtt{P}}(\vec{x}) \leq f(|\vec{x}|)$, and then we also have $\#_{\mathtt{P}}(\vec{x}) \leq q(|\vec{x}|)$ for some polynomial q since every function in $\mathcal{E}^2$ is bounded by a polynomial. So, the number of steps in the execution of the stack program P is bounded by a polynomial in the length of the input. It is a standard exercise to prove that P can be simulated by a Turing machine also working in polynomial time. It follows that every function computed by P is in PF.

Assume that every function computed by P is in PF. Then we have $\#_{\mathtt{P}}(|\vec{x}|) \leq q(|\vec{x}|)$ for some polynomial q. (Assume that such a q does not exists. Then then at least one loop in P will be controlled by a stack which length is not bounded by any polynomial in the length of the input. But then, since S_0-programs are length-monotonic, the program P computes a function which is not in PF.) Every polynomial is bounded by a function in $\mathcal{E}^2$. Hence, P has μ-measure 0 by Theorem 7.3. □

Definition 7.5. We shall define the measure ν for the programming languages S_1 and S. The definition is analogous to the definition of the ν-measure for loop programs.

Let $\prec_{\mathtt{P}}$ be defined as in Definition 7.2. Let $\mathtt{X:=_PY}$ denote that X:= Y is a subprogram of P. The relation $\xrightarrow{\mathtt{P}}$ is the smallest relation such that

- if $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$, then $\mathtt{X} \xrightarrow{\mathtt{P}} \mathtt{Y}$
- $\xrightarrow{\mathtt{P}}$ is transitive
- if $\mathtt{X:=_PY}$ and $\mathtt{Z} \xrightarrow{\mathtt{P}} \mathtt{Y}$, then $\mathtt{Z} \xrightarrow{\mathtt{P}} \mathtt{X}$

Then we define *clique*, *cover set*, etcetera exactly as above, and finally we define the ν-measure $\nu(\mathtt{P})$ of the program $\mathtt{P}$ by

- $\nu(\mathtt{imp}) = 0$ for every imperative $\mathtt{imp}$.
- Let $\mathtt{P} \equiv \mathtt{Q}_1\,;\mathtt{Q}_2$. Then $\nu(\mathtt{P}) = \max(\nu(\mathtt{Q}_1), \nu(\mathtt{Q}_2))$.
- Let $\mathtt{P} \equiv \mathtt{if\ top(X)} \equiv \mathtt{a\ [Q]}$. Then $\nu(\mathtt{P}) = \nu(\mathtt{Q})$.
- Let $\mathtt{P} \equiv \mathtt{foreach\ X\ [Q]}$. Then

$$\nu(\mathtt{P}) = \begin{cases} \nu(\mathtt{Q}) + 1 & \text{if } \mathtt{Q} \text{ has a clique of degree } \nu(\mathtt{Q}) + 1 \\ \nu(\mathtt{Q}) & \text{else} \end{cases}$$

End of definition.

Theorem 7.6 (Soundness of (ν, S)). We have

$$\nu(\mathtt{P}) \leq n \quad \Rightarrow \#_{\mathtt{P}}(\vec{x}) \leq f(|\vec{x}|) \text{ for some } f \in \mathcal{E}^{n+2}.$$

for every $n \in \mathbb{N}$ and every $\mathtt{P} \in S_1$.

Proof. Analogous to the proof of Theorem 6.1. Introduce a language S_M with nice monotonicity properties such that for every program $\mathtt{P} \in S$ there exists a program $\mathtt{Q} \in S_M$ such that $\nu(\mathtt{P}) = \nu(\mathtt{Q})$ and $\mathtt{P} \ll \mathtt{Q}$. Prove a flattening lemma for S_M, i.e. a lemma corresponding to Lemma 5.5. Thereafter proceed as in the proof of Theorem 6.1. □

To be able to state adequacy results for stack programs we introduce a new hierarchy.

Definition 7.7. For every $n \in \mathbb{N}$ we define $\mathcal{H}^n$ to be the class of functions from tuples of strings over Σ into strings over Σ which can be computed by a Turing machine which works in time f for some $f \in \mathcal{E}^n$, i.e. the number of steps in the execution of the Turing machine on input w is bounded by $f(|w|)$ for some $f \in \mathcal{E}^2$. (Now, $|w|$ denotes as usual the length of the string w, so $\mathcal{H}^2 = \text{PF}$.) *End of definition.*

Theorem 7.8 (Adequacy of (ν, S)). Every function in $\mathcal{H}^{n+2}$ can be computed by a S-program with ν-measure n (for every $n \in \mathbb{N}$).

Proof. We skip this proof. A proof of a similar result can be found in Kristiansen-Niggl [13]. □

Corollary 7.9. (i) A function is in PF if and only if it can be computed by an S-program with ν-measure 0. (ii) A problem is in P if and only if it can be decided by a program with ν-measure 0.

Proof. Recall that $\mathcal{H}^2 = \text{PF}$. Hence, the corollary follows straightaway from Theorem 7.6 and Theorem 7.8. □

8 Subelementary complexity classes

In this section we will focus on some subclasses of the Kalmár elementary functions. In order to have a suitable framework for our analysis we will introduce a modified version of the Grzegorczyk hierarchy. In this hierarchy we have the Kalmár elementary relations on level 4; we have PSPACE on level 3; we have LINSPACE on level 2, 1 and 0.

Definition 8.1. We define the sequence $\{G_i\}_{i\in\mathbb{N}}$ of number-theoretic functions by $G_0(x) = x + 1$, $G_1(x) = 2x + 1$, $G_2 = x^2 + 2$, $G_3(x) = 2^{(|x|^2)}$, $G_4(x) = 2^x$, and $G_{i+1}(x) = G_i^x(2)$ for $i \geq 4$. (The function $|x|$ is usually defined as the number bits of required to represent the number x, i.e. $|x| = \lceil \log_2 x \rceil$.) The *ith modified Grzegorczyk class* $\mathcal{G}^i$, is the least class of functions containing the initial functions zero, successor, projections, maximum and G_i, and is closed under composition and bounded simultaneous recursion. We dub $\{\mathcal{G}^i\}_{i\in\mathbb{N}}$ the *modified Grzegorczyk hierarchy. End of definition.*

Theorem 8.2. (i) For every $n \in \mathbb{N}$ and $f \in \mathcal{G}^n$ there exists a fixed number k such that $f(\vec{x}) \leq G_n^k(\max(\vec{x}))$. Thus, we have $\mathcal{G}^n \subset \mathcal{G}^{n+1}$ for any $n \in \mathbb{N}$. (ii) $\mathcal{E}^0 \subset \mathcal{G}^0$. (iii) $\mathcal{E}^2 = \mathcal{G}^2$, and thus $\mathcal{G}^2$ equals LINSPACEF. (iv) $\mathcal{G}^{n+1} = \mathcal{E}^n$ for $n \geq 3$. In particular, $\mathcal{G}^4$ equals the class of Kalmár-elementary functions. (v) $\mathcal{G}_\star^0 = \mathcal{G}_\star^1 = \mathcal{G}_\star^2$. Thus, each of these classes equals LINSPACE. (vi) $\mathcal{G}^3 =$ PSPACEF.

Proof. (i) Use induction on the definition of f. (ii) It is obvious that $\mathcal{E}^0 \subseteq \mathcal{G}^0$. Further, max is one of the initial functions in $\mathcal{G}^0$, but $\mathcal{E}^0$ does not contain max. To see this, assume $\max \in \mathcal{E}^0$. Then there exist constants k and $i \in \{1, 2\}$ such that $\max(x_1, x_2) \leq x_i + k$. This is a nonsense. (iii) and (iv).

It is obvious that $\mathcal{E}^2 \subseteq \mathcal{G}^2$ and that $\mathcal{E}^i \subseteq \mathcal{G}^{i+1}$ for $i \geq 3$. The right-to-left inclusions follows from the fact that $\mathcal{E}^i$ is closed under bounded simultaneous recursion for all $i \geq 2$. (v) Muchnick [24] studies the vectorised Grzegorczyk hierarchy $\mathcal{E}^{v,0}, \mathcal{E}^{v,1}, \mathcal{E}^{v,2}, \ldots$. He proves that $\mathcal{E}^{v,i}_\star = \mathcal{E}^2_\star$ for $i = 0, 1, 2$. Thus (v) holds since it is obvious that $\mathcal{E}^{v,i}_\star \subseteq \mathcal{G}^i_\star$. (vi) We skip this proof. □

Note 8.3. (i) We would have achieved the same hierarchy if we had defined $G_4(x) = G^x_3(2)$. Thus, we could have defined the backbone functions in a uniform way from level 4 and upwards. Transparency is the sole motive for defining $G_4(x)$ to equal 2^x. (ii) The modified Grzegorczyk hierarchy is not an unnatural hierarchy compared to the original hierarchy. The class $\mathcal{G}^3$ is in a way artificially inserted into the hierarchy, but one should note, so is $\mathcal{E}^2$ in the original hierarchy. One application of of unbounded primitive recursion over functions in $\mathcal{E}^1$ might yield a function on the Kalmár-elementary level, i.e. a function which is not in $\mathcal{E}^2$. Thus, one could argue that $\mathcal{E}^2$ is artificially inserted into the the hierarchy. We cannot uniformly define the original hierarchy all the way from the very the bottom without loosing the LINSPACE-level. In the modified hierarchy we have in addition to the LINSPACE-level ($\mathcal{G}^2$) inserted a PSPACE-level ($\mathcal{G}^3$). One unbounded application of simultaneous (or primitive) recursion over functions in $\mathcal{G}_i$ for $i = 1, 2, 3$ might yield a function on the Kalmár-elementary level, i.e. on the 4th level of the hierarchy. (iii) The classes in the modified hierarchy retain all the closure properties of classes in the original hierarchy, and the class $\mathcal{G}^3$ is no exception. (More on generalised Grzegorczyk classes can be found in Kristiansen [12].) (iv) The classes $\mathcal{G}^0$ and $\mathcal{G}^1$ are by definition closed under bounded simultaneous recursion, whereas it is an open problem whether $\mathcal{E}^0$ and $\mathcal{E}^1$ are closed under such recursion. Thus it also becomes an open problem if $\mathcal{G}^1 = \mathcal{E}^1$. *End of note.*

Definition 8.4. Let L be defined as in the previous sections, i.e. as the loop language with the imperatives `suc(X)`, `X:= Y`, `pred(X)` and `nil(X)`. Let $L^\emptyset$ be the set of L programs `P` such that the relation $\xrightarrow{\texttt{P}}$ is empty, and let L^{IR} be the set of L programs `P` such that the relation $\xrightarrow{\texttt{P}}$ is irreflexive. Let L^n be the set of programs with ν-measure n. If $L^\bullet$ is a set of loop programs, then $\mathcal{L}^\bullet$ denotes the set of functions which can be computed by the programs in $L^\bullet$. *End of definition.*

Theorem 8.5. $\mathcal{L}^{\text{IR}}_\star = \mathcal{L}^0_\star = \mathcal{G}^2_\star = \text{LINSPACE}$.

Proof. It is quite obvious that $\mathcal{L}^{\mathrm{IR}} = \mathcal{L}^0$. It follows from 6.2 that $\mathcal{L}^0 = \mathcal{E}^2 =$ LINSPACEF. Theorem 8.2 says that $\mathcal{G}^2 = \mathcal{E}^2$. □

Lemma 8.6. Let P be an L-program where $\mathcal{V}(\mathtt{P}) = \vec{\mathtt{X}}$. Let m be a fixed number such that no register during an execution of P on input $\vec{\mathtt{X}} = \vec{x}$ exceeds m. Then there exists $\mathtt{Q} \in L^{\emptyset}$ such that

$$\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{x}'\} \quad \Leftrightarrow \quad \{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} = m\}\mathtt{Q}\{\vec{\mathtt{X}} = \vec{x}'\} \ .$$

Proof. Let U and V be a fresh variables. Let P′ be P where every subprogram on the form `pred(Z)` is replaced by the subprogram

$$\mathtt{nil(U)\,;\ loop\ Z\ [\ V:=\ U;suc(U)\ ]\,;\ Z:=\ V}\ .$$

Then we have $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{y}\}$ iff $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}'\{\vec{\mathtt{X}} = \vec{y}\}$ and there are no occurrences of `pred(..)` in P′. Now, let M be a fresh variable, let the function τ from L-programs (with no occurrences of `pred(..)`) into L-programs be defined by

- $\tau(\mathtt{P\,;\ Q}) = \tau(\mathtt{P})\,;\ \tau(\mathtt{Q})$
- $\tau(\mathtt{loop\ W\ [P]}) = \mathtt{loop\ \ W\ \ [}\tau(\mathtt{P})\mathtt{]}$
- $\tau(\mathtt{W:=\ Y}) = \tau(\mathtt{W:=\ Y})$
- $\tau(\mathtt{nil(W)}) = \mathtt{W:=\ M}$
- $\tau(\mathtt{suc(W)}) = \mathtt{pred(W)}$

and let $\mathtt{P}'' = \tau(\mathtt{P}')$. The program P″ has no occurrences of the statement `suc(..)`, and for all sufficiently large m we have $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{y}\}$ iff $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} = m\}\mathtt{P}''\{\mathtt{X}_1 = m - y_1, \ldots, \mathtt{X}_l = m - y_l\}$ where $\vec{\mathtt{X}} = \mathtt{X}_1, \ldots, \mathtt{X}_l$. Let U be a fresh variable and let

$$\mathtt{R}_i \equiv \mathtt{U:=\ M;\ loop\ X}_i\mathtt{[pred(U)]\,;\ X}_i\mathtt{:=\ U}$$

Finally, let

$$\mathtt{Q} \ \equiv \ \mathtt{P''\,;\ R}_1\mathtt{\,;\ \ldots\ ;\ R}_l\ .$$

Then, we have $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{x}'\}$ iff $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} = m\}\mathtt{Q}\{\vec{\mathtt{X}} = \vec{x}'\}$. □

Lemma 8.7. The function $\max(x, y)$ can be computed by a program in $L^{\emptyset}$.

Proof. Let P be the program

```
U:= X; loop Y [pred(U)];
nil(V); suc(V); loop U [pred(V)];
Z:= X; loop V [Z:= Y]
```

Then P is in $L^{\emptyset}$ and $\{\mathtt{X} = x, \mathtt{Y} = y\}\mathtt{P}\{\mathtt{Z} = \max(x, y)\}$. □

Theorem 8.8. $\mathcal{G}^0 = \mathcal{L}^{\emptyset}$.

Proof. First we prove $\mathcal{G}^0 \subseteq \mathcal{L}^{\emptyset}$. Assume $f \in \mathcal{G}^0$. It is easy to see that there exists an L-program P such that $\{\vec{\mathtt{Z}} = \vec{z}\}\mathtt{P}\{\mathtt{Y} = f(\vec{z})\}$. It is also easy to see that there is fixed number k such that no register exceeds the value $\max(\vec{z})+k$ during an execution of P on input $\vec{\mathtt{Z}} = \vec{z}$. Lemma 8.7 entails that there exists a program `imax` in $L^{\emptyset}$ such that $\{\vec{\mathtt{X}} = \vec{x}\}$ `imax` $\{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} = \max(\vec{x}) + k\}$, and then Lemma 8.6 entails that f can be computed by a program in $L^{\emptyset}$. This completes the proof of $\mathcal{G}^0 \subseteq \mathcal{L}^{\emptyset}$.

The proof of $\mathcal{L}^{\emptyset} \subseteq \mathcal{G}^0$ is straightforward. Let $L^{\emptyset-}$ be the set of $L^{\emptyset}$ programs with no occurrence of imperatives on the form `suc(X)`. Use induction over the syntax of programs to prove that for each $\mathtt{P} \in L^{\emptyset-}$ where $\mathcal{V}(\mathtt{P}) = \{\mathtt{X}_1, \ldots, \mathtt{X}_n\} = \vec{\mathtt{X}}$ there exists functions $f_1, \ldots, f_n \in \mathcal{G}^0$ such that

$$\{\vec{\mathtt{X}} = \vec{x}\}\ \mathtt{P}\ \{\mathtt{X}_1 = f_1(\vec{x}) \leq \max(\vec{x}), \ldots, \mathtt{X}_1 = f_n(\vec{x}) \leq \max(\vec{x})\}\ .$$

The desired result follows easily. □

Corollary 8.9. $\mathcal{L}^{\emptyset}_{\star} = \mathcal{L}^{\mathrm{IR}}_{\star} = \textsc{linspace}$.

Proof. The equalities follow from the theorems 8.5, 8.8 and 8.2. □

Note 8.10. The previous corollary implies that we cannot characterise any smaller complexity class than LINSPACE solely by imposing restrictions on the relation "X controls Y" in any language containing L.

Definition 8.11. Recall that S denotes the set of stack programs (defined in a previous section). Let $S^{\emptyset}$ be the set of S-programs P such that the relation $\xrightarrow{\mathtt{P}}$ is empty, and let S^{IR} be the set of S-programs P such that the relation $\xrightarrow{\mathtt{P}}$ is irreflexive. Let S^n be the set of program with ν-measure n. If $S^{\bullet}$ is a set stack of programs, then $\mathcal{S}^{\bullet}$ denotes the set of functions which can be computed by programs in $S^{\bullet}$. *End of definition.*

Theorem 8.12. $\mathcal{S}^0_\star = \mathcal{S}^{\mathrm{IR}}_\star = \mathrm{P}$.

Proof. Corollary 7.9 states that $\mathcal{S}^0_\star = \mathrm{P}$. It is easy to prove that $\mathcal{S}^0_\star = \mathcal{S}^{\mathrm{IR}}_\star$. □

We have seen that $\mathcal{L}^\emptyset_\star = \mathcal{L}^{\mathrm{IR}}_\star$. In contrast, the next theorem tells that $\mathcal{S}^\emptyset_\star$ is strictly included in $\mathcal{S}^{\mathrm{IR}}_\star$.

Theorem 8.13. $\mathrm{CONSPACE} \subset \mathcal{S}^\emptyset_\star \subset \mathcal{S}^{\mathrm{IR}}_\star = \mathrm{P}$.

Proof. Every one-way finite automaton can be simulated by a program in $S^\emptyset$. Thus, $\mathrm{CONSPACE} \subseteq \mathcal{S}^\emptyset_\star$. (CONSPACE equals the class of languages recognised by such automatons, see Odifreddi [26].) Let

$$A \;=\; \{w \mid\; |w| = x^2 \text{ for some } x \in \mathbb{N}\}\ .$$

Membership in A can be decided by a program in $S^\emptyset$, but not by a finite automaton. Hence $\mathrm{CONSPACE} \subset \mathcal{S}^\emptyset_\star$. It is trivial that $\mathcal{S}^\emptyset_\star \subseteq \mathcal{S}^{\mathrm{IR}}_\star$. Let w^R denote the word w reversed. Let $B = \{w \mid\; w = w^R\}$. Membership in B can obviously be decided by a Turing machine working in polynomial time, but no program in $\mathcal{S}^\emptyset$ can decide membership in B. Thus $\mathcal{S}^\emptyset_\star \subset \mathcal{S}^{\mathrm{IR}}_\star = \mathrm{P}$. □

The languages L and S can be merged into one imperative programming language I. The resulting language I computes both on numbers and on symbols in the alphabet $\{0, 1\}$. (All our result can be generalised to arbitrary alphabets with cardinality ≥ 2.) Still, the language will have only one type of variables. Whether a variable `X` hold a natural number or a stack over the fixed alphabet $\{0, 1\}$ depends on the point of view. Used in an L-construction, e.g. `suc(X)`, we view `X` as a number variable; used in an S-construction, e.g. `push(0,X)`, we view `X` as a stack variable. In order to make this strategy work we need a suitable bijection between the natural numbers an the strings over the alphabet $\{0, 1\}$.

Definition 8.14. We use $\mathbb{W}$ to denote the set of *words*, i.e. the set of strings over bits (the alphabet $\{0, 1\}$). We use ε to denote the empty word. As usual, $|w|$ denotes the length of the word w, and w_i denotes the ith bit of the word w starting from 0 in the rightmost position. Thus, if $|w| = 4$, then $w = w_3, w_2, w_1, w_0$. We use juxtaposition to concatenate words. So, e.g. $w0$

denotes the word w extendend by 0 in the rightmost position. The function $\sigma : \mathbb{W} \to \mathbb{N}$ is defined by

$$\sigma(w) = 2^n + w_{n-1}2^{n-1} + \cdots + w_1 2^1 + w_0 2^0 - 1$$

where $n = |w|$. *End of definition.*

The function σ is a bijection and the numbers $0, 1, 2, 3, \ldots$ are respectively mapped to the words $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, \ldots$.

Lemma 8.15. (1) σ is a bijection. (2) $\sigma(w) \leq 2^{|w|+1}$. (3) $\sigma(w0) = 2(\sigma(w) + 1) - 1$ and $\sigma(w1) = 2(\sigma(w) + 1)$.

Proof. We leave (1) and (2) to the reader. Let $n = |w|$. Then

$$\begin{array}{lcll} \sigma(w0) & = & 2^{n+1} + w_{n-1}2^n + \cdots + w_0 2^1 + (0 \times 2^0) - 1 & \text{def. of } \sigma(w0) \\ & = & 2(2^n + w_{n-1}2^{n-1} + \cdots + w_0 2^0) - 1 & \\ & = & 2(2^n + w_{n-1}2^{n-1} + \cdots + w_0 2^0 - 1 + 1) - 1 & \\ & = & 2(\sigma(w) + 1) - 1 & \text{def. of } \sigma(w) \end{array}$$

This proves $\sigma(w0) = 2(\sigma(w) + 1) - 1$. A similar argument proves $\sigma(w1) = 2(\sigma(w) + 1)$ □

Note 8.16. We push and pop bits on the right hand side of a word, e.g. $\{\mathtt{X} = w\}\mathtt{push(0,X)}\{\mathtt{X} = w0\}$ and $\{\mathtt{X} = w1\}\mathtt{pop(X)}\{\mathtt{X} = w\}$.

Definition 8.17 *(General imperative programs).* The syntax of the programming language I are inductively defined as follows:

- Every *imperative* among `nil(X)`, `suc(X)`, `pred(X)`, `pop(X)`, `push(b,X)` `X:= Y` is an program (for any variables `X`, `Y` and `b` $\in \{0, 1\}$) .
- If `P` is a program with no occurrence of the variable `X` in an imperative, then so is the *loop* `loop X [P]` (for any variables `X`).
- If `P` is a program, then so is every *conditional* `if top(X)` $\equiv$ `b [P]` (for every variable `X` and `b` $\in \{0, 1\}$).
- If `P` is a program with no occurrence of the variable `X` in an imperative, then so is the *loop* `foreach X [P]` (for any variables `X`).
- If $\mathtt{P}_1, \mathtt{P}_2$ are programs, then so is the *sequence* $\mathtt{P}_1\mathtt{;}\ \mathtt{P}_2$.

The semantics of I are a straightforward merging of the semantics of the languages L and S. Note that since $\sigma(\varepsilon) = 0$, the imperative `nil(X)` will turn `X` into the empty stack when `X` is viewed as an stack, and set `X` to 0 if `X` is viewed as a natural number. *End of definition.*

Example 8.18. The following program computes the function G_4. (Recall that $G_4(x) = 2^x$.)

$$\{\mathtt{X} = x\}\ \mathtt{nil(Y);\ loop\ X\ [push(0,Y)];\ suc(Y)}\{\mathtt{Y} = 2^x\}\ . \qquad (*)$$

The next program computes the length function $|w|$.

$$\{\mathtt{X} = w\}\ \mathtt{nil(Y);\ foreach\ X\ [suc(Y)]}\{\mathtt{Y} = |w|\}\ . \qquad (**)$$

End of example.

Definition 8.19. We define relation $\xrightarrow{\mathtt{P}}$ for programs in I. (The definition is analogous to the definition of the corresponding relations for programs in L and S.)

The relations $\prec_{\mathtt{P}}$ and $\xrightarrow{\mathtt{P}}$ are binary relations over $\mathcal{V}(\mathtt{P})$. The relation $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$ holds iff at least one of (i) and (ii) holds:

(i) `P` has a subprogram `loop X [Q]` where `suc(Y)` or `push(b,Y)` are subprograms of `Q`.

(ii) `P` has a subprogram `foreach X [Q]` where `suc(Y)` or `push(b,Y)` are subprograms of `Q`.

Let $\mathtt{X{:=}_P Y}$ denote that `X:= Y` is a subprogram of `P`. The relation $\xrightarrow{\mathtt{P}}$ is the smallest relation such that

- if $\mathtt{X} \prec_{\mathtt{P}} \mathtt{Y}$, then $\mathtt{X} \xrightarrow{\mathtt{P}} \mathtt{Y}$
- $\xrightarrow{\mathtt{P}}$ is transitive
- if $\mathtt{X{:=}_P Y}$ and $\mathtt{Z} \xrightarrow{\mathtt{P}} \mathtt{Y}$, then $\mathtt{Z} \xrightarrow{\mathtt{P}} \mathtt{X}$

Let $I^{\emptyset}$ denote the set of I-programs `P` where the relation $\xrightarrow{\mathtt{P}}$ is empty, and let I^{IR} denote the set of I-programs `P` where the relation $\xrightarrow{\mathtt{P}}$ is irreflexive. Let

$I^{\text{IR-}}$ denote the set of I-programs P such that $\texttt{P} \in I^{\text{IR}}$ and no subprogram of P on the form `loop X [Q]` has a subprogram on the form `push(b,Y)`. If $I^\bullet$ is a set of imperative programs, then $\mathcal{I}^\bullet$ denotes the class of functions which can be computed by the programs in $I^\bullet$. *End of definition.*

Example 8.20. The program (**) in Example 8.18 is in $I^{\text{IR-}}$. The program (*) in the same example is in I^{IR}, but not in $I^{\text{IR-}}$. The following program which computes the function G_3 is in $I^{\text{IR-}}$. (Recall that $G_3(x) = 2^{|x|^2}$, and if 0^n denotes a string of n zeros, then $\sigma(0^n) = 2^n - 1$.)

$\{\texttt{X} = x\}$
`nil(Y); Z:= X; foreach X [foreach Z [push(0,Y)]]; suc(Y)`
$\{\texttt{Y} = 2^{|x|^2}\}$

End of example.

Theorem 8.21. The class $\mathcal{I}^{\text{IR}}$ equals the class of Kalmàr elementary functions $\mathcal{G}^4$.

Proof. Use induction on the syntactical build-up of a program $\texttt{P} \in I^{\text{IR}}$ to prove that for every function f computed by P we have $f(\vec{x}) \leq G_4^k(\max(\vec{x}))$ for some fixed number k. It follows easily that $\mathcal{I}^{\text{IR}} \subseteq \mathcal{G}^4$ since $\mathcal{G}^4$ is closed under composition and bounded simultaneous recursion.

Assume $f \in \mathcal{G}^4$. Use induction on a definition of f to prove that $f(\vec{x})$ can be computed by a program $\texttt{P} \in L$ such that during the computation no register exceeds the value $G_4^k(\max(\vec{x}))$ for some fixed number k. By Lemma 8.6 there is a program $\texttt{Q} \in L^{\emptyset}$ (and thus $\texttt{Q} \in I^{\text{IR}}$) such that

$$\{\vec{\texttt{X}} = \vec{x}\}\texttt{P}\{\vec{\texttt{X}} = \vec{x}'\} \quad \Leftrightarrow \quad \{\vec{\texttt{X}} = \vec{x}, \texttt{M} \geq G_4^k(\max(\vec{x}))\}\texttt{Q}\{\vec{\texttt{X}} = \vec{x}'\} \ .$$

Example 8.18 shows that the function G_4 can be computed by a program in $\mathcal{I}^{\text{IR}}$. Hence, a program in $\mathcal{I}^{\text{IR}}$ can also compute the function $G_4^k(\max(\vec{x}))$. It follows that f can be computed by a program in I^{IR}. Thus we have proved $\mathcal{G}^4 \subseteq \mathcal{I}^{\text{IR}}$. □

Theorem 8.22. The class $\mathcal{I}^{\text{IR-}}$ equals the class of polynomial space computable functions $\mathcal{G}^3$, i.e. PSPACEF.

Proof. This proof is identical to the proof of Theorem 8.21 where "4" is replaced by "3", "IR" is replaced by "IR-" and "Example 8.18" is replaced by "Example 8.20". □

In order to state some normal form results, we shall introduce the notion of a *core language* and *core programs*. Roughly speaking, the core language is the part of the programming language we need to compute fast growing functions.

Definition 8.23. The set of *core programs* is a subset of the set of I-programs. A *core program* is defined by

- every *imperative* among `suc(X)`, `push(0,X)`, `push(1,X)` is a core program (for any variable `X`).
- If `P` is a core program with no occurrence of the variable `X` in an imperative, then so are `loop X [P]` and `foreach X [P]` (for any variables `X`).
- If $\mathtt{P}_1, \mathtt{P}_2$ are core programs, then so is $\mathtt{P}_1\,\mathtt{;}\,\mathtt{P}_2$.

Assume $\mathcal{V}(\mathtt{P}) = \vec{\mathtt{X}}$. We us $!_{\mathtt{P}}(\vec{x})$ to denote the least natural number m such that no register exceeds m during an execution of the program `P` on input $\vec{\mathtt{X}} = \vec{x}$.

Assume $\mathcal{V}(\mathtt{P}) \subseteq \mathcal{V}(\mathtt{Q})$. Let us say, $\mathcal{V}(\mathtt{P}) = \vec{\mathtt{X}}$ and $\mathcal{V}(\mathtt{Q}) = \vec{\mathtt{X}}, \vec{\mathtt{Y}}$. We use $\mathtt{P} \sim \mathtt{Q}$ to denote that `Q` computes the same functions as `P` with respect to the variables $\vec{\mathtt{X}}$, i.e. the relation $\mathtt{P} \sim \mathtt{Q}$ holds if and only if

$$\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{z}\} \;\Leftrightarrow\; \{\vec{\mathtt{X}} = \vec{x}, \vec{\mathtt{Y}} = \vec{y}\}\mathtt{Q}\{\vec{\mathtt{X}} = \vec{z}\}\ .$$

End of definition.

Lemma 8.24. Let *Lan* be any of the programming languages $I^{\text{IR-}}$, I^{IR}, L^{IR}, L^n (for $n \in \mathbb{N}$), S^{IR}, and S^n (for $n \in \mathbb{N}$). Let $\mathcal{V}(\mathtt{P}) = \vec{\mathtt{X}}$. If $\mathtt{P} \in \mathit{Lan}$, then there exists a core program $\mathtt{Q} \in \mathit{Lan}$ such that $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{Q}\{\vec{\mathtt{X}} = \vec{x}, \mathtt{Z} \geq !_{\mathtt{P}}(\vec{x})\}$ (where `Z` is any fresh variable).

Proof. We leave this proof to the reader. □

Lemma 8.25. For each $\mathtt{P} \in I$ (and thus we might have $\mathtt{P} \in S$) there exists $\mathtt{Q} \in L$ such that $\mathtt{P} \sim \mathtt{Q}$ and $!_{\mathtt{Q}}(\vec{x}, 0, \ldots, 0) \leq !_{\mathtt{P}}(\vec{x}) + k$ for some fixed number k.

Proof. Let $\mathtt{Y}$ be a fresh variable. Lemma 8.15 says that $\sigma(s0) = 2(\sigma(s)+1)-1$. Thus, if $\mathtt{Q}_0$ is the program

```
suc(X); nil(Y); loop X [suc(Y); suc(Y)]; X:= Y; pred(X)
```

we have $\mathtt{push(0,X)} \sim \mathtt{Q}_0$. Furthermore, we

$$!_{\mathtt{Q}_0}(x, 0) \;\leq\; !_{\mathtt{push(0,X)}}(x) \;+\; 1\,.$$

Obviously, we also have $\mathtt{Q}_1 \in L$ such that $\mathtt{push(1,X)} \sim \mathtt{Q}_1$ and such that we have the required bound on $!_{\mathtt{Q}_1}$. Let $\mathtt{P}_0$ be $\mathtt{P}$ where each occurrence of $\mathtt{push(0,X)}$ is replaced by $\mathtt{Q}_0$ and each occurrence of $\mathtt{push(1,X)}$ is replaced by $\mathtt{Q}_1$. Then we have $\mathtt{P} \sim \mathtt{P}_0$ and $!_{\mathtt{P}_0}(\vec{x}, \vec{0}) \leq !_{\mathtt{P}}(x) + k$ for some fixed k.

Now we have a program $\mathtt{P}_0$ without "push". We can proceed in the same way to remove the occurrences of "pop" and the constructions on the form $\mathtt{if\ top(X)} \equiv \mathtt{a\ [R]}$ and $\mathtt{foreach\ X\ [R]}$. It is a rather straightforward process. □

Theorem 8.26 (Normal Form). Let *Lan* be any of the programming languages $I^{\text{IR-}}$, I^{IR}, L^{IR}, L^n (for $n \in \mathbb{N}$), S^{IR}, and S^n (for $n \in \mathbb{N}$). For any $\mathtt{P} \in \mathit{Lan}$ there exists a core program $\mathtt{C} \in \mathit{Lan}$ and a program $\mathtt{Q} \in L^{\emptyset}$ such that $\mathtt{P} \sim \mathtt{C;\ Q}$.

Proof. Assume $\mathtt{P} \in \mathit{Lan}$. By Lemma 8.25 there is an L-program $\mathtt{Q}_0$ such that $\mathtt{P} \sim \mathtt{Q}_0$ and $!_{\mathtt{Q}_0}(\vec{x}, \vec{0}) \leq !_{\mathtt{P}}(\vec{x}) + k$ for some fixed k. Then, by Lemma 8.6 there exists $\mathtt{Q} \in L^{\emptyset}$ such that

$$\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{x}'\} \;\;\Leftrightarrow\;\; \{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} = m\}\mathtt{Q}\{\vec{\mathtt{X}} = \vec{x}'\}$$

whenever $m \geq !_{\mathtt{P}}(\vec{x}) + k$. By Lemma 8.24 there exists a core program $\mathtt{C} \in \mathit{Lan}$ such that $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{C}\{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} \geq !_{\mathtt{P}}(\vec{x}) + k\}$. Hence

$$\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{P}\{\vec{\mathtt{X}} = \vec{x}'\} \;\;\Leftrightarrow\;\; \{\vec{\mathtt{X}} = \vec{x}\}\mathtt{C;\ Q}\{\vec{\mathtt{X}} = \vec{x}'\}$$

i.e. we have $\mathtt{P} \sim \mathtt{C;\ Q}$, where $\mathtt{Q} \in L^{\emptyset}$ and $\mathtt{C}$ is a core program in *Lan*. □

Corollary 8.27 (Normal Form). For every program $\mathtt{P} \in I^{\text{IR-}}$ there are programs $\mathtt{Q}_0 \in S^{\text{IR}}$ and $\mathtt{Q}_1 \in L^{\emptyset}$ such that $\mathtt{P} \sim \mathtt{Q}_0\,;\mathtt{Q}_1$.

Proof. By Theorem 8.26 there exists a core program $\mathtt{C} \in I^{\text{IR-}}$ and a program $\mathtt{Q}_1 \in L^{\emptyset}$ such that $\mathtt{P} \sim \mathtt{C};\mathtt{Q}_1$. The proof of the theorem shows that $\mathtt{C}$ has the property $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{C}\{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} \geq !_{\mathtt{P}}(\vec{x}) + k\}$ and that the theorem will hold for any program $\mathtt{C}'$ with this property. Since $\mathtt{C} \in I^{\text{IR-}}$, there will be a program in S^{IR} satisfying the property, i.e. there is a program $\mathtt{Q}_0 \in \mathtt{S}^{\text{IR}}$ such that $\{\vec{\mathtt{X}} = \vec{x}\}\mathtt{Q}_0\{\vec{\mathtt{X}} = \vec{x}, \mathtt{M} \geq !_{\mathtt{P}}(\vec{x}) + k\}$. Hence, we have also have $\mathtt{P} \sim \mathtt{Q}_0;\mathtt{Q}_1$ where $\mathtt{Q}_0 \in S^{\text{IR}}$ and $\mathtt{Q}_1 \in L^{\emptyset}$. □

Corollary 8.28. If P ≠ PSPACE, then LINSPACE \ P ≠ ∅.

Proof. We assume LINSPACE \ P = ∅, and thus $\mathcal{L}^{\emptyset}$ = LINSPACE ⊆ P = $\mathcal{S}^{\text{IR}}$, and prove that P = PSPACE.

Pick an arbitrary problem α in PSPACE. Now, PSPACE = $\mathcal{I}_{\star}^{\text{IR-}}$ and thus there is a program $\mathtt{P} \in I^{\text{IR-}}$ which solves the problem. According to Corollary 8.27 there are programs $\mathtt{Q} \in S^{\text{IR}}$ and $\mathtt{R} \in L^{\emptyset}$ such that $\mathtt{P} \sim \mathtt{Q};\mathtt{R}$. Now, since we have LINSPACE ⊆ P by our assumption, every zero-one function computed by $\mathtt{R}$ can be computed by a program in S^{IR}. Hence, there is a program $\mathtt{P}' \in S^{\text{IR}}$ which solves the problem α, and thus $\alpha \in$ P. This proves PSPACE ⊆ P. The inclusion P ⊆ PSPACE is trivial. Hence P = PSPACE. □

Corollary 8.28 is of course a very well known fact. Indeed, many problems in LINSPACE are known to be PSPACE complete. Still, it is nice to see that such a corollary follows from our theory on programming languages.

9 Uniform translations between formalisms

Several formalisms have been introduced in order to give implicit characterisations of complexity classes, e.g. predicative recursion (Bellantoni-Cook [1]), tiered definition schemes (Leivant [17]), measures on lambda terms (Niggl [25]) and measures on primitive recursive definitions (Bellantoni-Niggl [5]). In contrast to these formalism, the imperative languages introduced in this report are flexible programming languages closely related to a von Neumann computer architecture. It is likely that programs and definitions of functions in several other formalism easily and uniformly can be translated into our imperative languages. Theorem 9.2 exemplifies the type of translation results we are discussing. We consider the theorem to be a mere exercise preparing us for more sophisticated results along the same line.

There can be several benefits of discovering such translations. It might yield a valuable analysis of actual computations of functions defined in a formalism, e.g. an analysis of the flow of data, or a refined analysis of the space requirements. Another benefit might be an easy and transparent proof that functions defined in a certain formalism is in a certain complexity class, e.g. it is a corollary of Theorem 9.2 below that the set of functions defined by predicative recursion over natural numbers is included in LINSPACEF. The theorem states that every function defined by such recursion can be computed by a program in L^{IR}, and $\mathcal{L}^{\mathrm{IR}}$ equals LINSPACEF.

Definition 9.1. Let B be the Bellantoni-Cook class of function definitions over $\mathbb{N}$ generated by the definition schemes

- *Safe composition:* $f(\vec{x};\vec{a}) = h(\vec{r}(\vec{x};);\vec{t}(\vec{x};\vec{a}))$
- *Predicative recursion on numbers:*

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

from the initial functions 0 (constant), $\pi_j^{m,n}(x_1,\ldots,x_n;x_{n+1},\ldots,x_{n+m})$ (projections), $s(;a)$ (successor), $p(;a)$ (predecessor) and $c(;a,b,c)$ (conditional, $c(;a,b,c) = b$ if $a = 0$ and $c(;a,b,c) = c$ if $a \neq 0$).

We say that an element in a partial ordering is *minimal* if it has no predecessors, and *maximal* if it has no successors. *End of definition.*

Theorem 9.2. For every $f \in B$ we can uniformly construct an L-program $\mathtt{P}$ such that

$$\{\vec{\mathtt{X}} = \vec{x}, \vec{\mathtt{A}} = \vec{a}\}\ \mathtt{P}\ \{\mathtt{O} = f(\vec{x};\vec{a})\}$$

and the relation $\xrightarrow{\mathtt{P}}$ is irreflexive (and thus an ordering relation). Moreover, the variables $\vec{\mathtt{X}}$ are minimal elements in the ordering $\xrightarrow{\mathtt{P}}$, and the variables $\vec{\mathtt{A}}$ and $\mathtt{O}$ are maximal elements.

Proof. We use induction on a definition of f to prove the theorem strengthened by the assertion

(*) the program $\mathtt{P}$ does not contain the subprogram $\mathtt{suc(Z)}$ for any $\mathtt{Z} \in \vec{\mathtt{X}}$.

The following programs shows that the theorem and (*) hold when f is an initial function.

- $\texttt{nil(O)}\{\texttt{O} = 0\}$
- $\{\texttt{A} = a\}\ \texttt{suc(A); O:=A}\ \{\texttt{O} = s(;a)\}$
- $\{\texttt{A} = a\}\ \texttt{pred(A); O:=A}\ \{\texttt{O} = p(;a)\}$
- $\{\vec{\texttt{X}} = \vec{x}\}\ \texttt{O:=X}_j\ \{\texttt{O} = \pi_j^{m,n}(x_1, \ldots, x_n; x_{n+1}, \ldots, x_{n+m})\}$
- $\{\texttt{A} = a, \texttt{B} = b, \texttt{C} = c\}\ \texttt{O:= B; for A [O:= C]}\ \{\texttt{O} = c(;a,b,c)\}$

Suppose f is defined by safe composition. It should be sufficient to carry out the proof for the case

$$f(x,y;a,b) = h(r_1(x,y;), r_2(x,y;); t_1(x,y;a,b), t_2(x,y;a,b))$$

. The induction hypothesis gives programs $\texttt{R}_1, \texttt{R}_2, \texttt{T}_1, \texttt{T}_2$ such that

$$\{\texttt{X}^{\texttt{R}_i} = x, \texttt{Y}^{\texttt{R}_i} = y\}\ \texttt{R}_i\ \{\texttt{O}^{\texttt{R}_i} = r_i(x,y;)\}$$

and

$$\{\texttt{X}^{\texttt{T}_i} = x, \texttt{Y}^{\texttt{T}_i} = y, \texttt{A}^{\texttt{T}_i} = a, \texttt{B}^{\texttt{T}_i} = b\}\ \texttt{T}_i\ \{\texttt{O}^{\texttt{T}_i} = t_i(x,y;a,b)\}$$

for $i = 1, 2$. Furthermore, the induction hypothesis yields a program $\texttt{H}$ such that

$$\{\texttt{X}^{\texttt{H}} = x, \texttt{Y}^{\texttt{H}} = y, \texttt{A}^{\texttt{H}} = a, \texttt{B}^{\texttt{H}} = b\}\ \texttt{H}\ \{\texttt{O}^{\texttt{H}} = h(x,y;a,b)\}\ .$$

We have renamed variables such that each variable occur in at most one of the programs. (The notation $\texttt{X}^{\texttt{Q}}$ indicates that the variable $\texttt{X}^{\texttt{Q}}$ does not occur in any other program than $\texttt{Q}$.) Let $\texttt{X}, \texttt{Y}, \texttt{A}, \texttt{B}, \texttt{O}$ be fresh variables. The theorem and (*) hold when $\texttt{P}$ is the program

$$\begin{array}{l}
\texttt{X}^{\texttt{R}_1}\texttt{:= X; Y}^{\texttt{R}_1}\texttt{:= Y; R}_1\texttt{; X}^{\texttt{R}_2}\texttt{:= X; Y}^{\texttt{R}_2}\texttt{:= Y; R}_2\texttt{;} \\
\texttt{X}^{\texttt{T}_1}\texttt{:= X; Y}^{\texttt{T}_1}\texttt{:= Y; A}^{\texttt{T}_1}\texttt{:= A; B}^{\texttt{T}_1}\texttt{:= B; T}_1\texttt{;} \\
\texttt{X}^{\texttt{T}_2}\texttt{:= X; Y}^{\texttt{T}_2}\texttt{:= Y; A}^{\texttt{T}_2}\texttt{:= A; B}^{\texttt{T}_2}\texttt{:= B; T}_2\texttt{;} \\
\texttt{X}^{\texttt{H}}\texttt{:= O}^{\texttt{R}_1}\texttt{; Y}^{\texttt{H}}\texttt{:= O}^{\texttt{R}_2}\texttt{; A}^{\texttt{H}}\texttt{:= O}^{\texttt{T}_1}\texttt{; B}^{\texttt{H}}\texttt{:= O}^{\texttt{T}_2}\texttt{; H; O:= O}^{\texttt{H}}
\end{array}$$

Suppose $f(0, \vec{x}; \vec{a}) = g(\vec{x}; \vec{a})$ and $f(y+1, \vec{x}; \vec{a}) = h(y, \vec{x}; \vec{a}, f(y, \vec{x}; \vec{a}))$. In order to avoid cluttered notation, we say that $\vec{x}$ is the single variable x and that $\vec{a}$ is the single variable a. The induction hypothesis yields programs $\texttt{G}$ and $\texttt{H}$ such that

$$\{\texttt{X}^{\texttt{G}} = x, \texttt{A}^{\texttt{G}} = a, \}\ \texttt{G}\ \{\texttt{O}^{\texttt{G}} = g(x;a)\}$$

and

$$\{\mathtt{Y}^{\mathtt{H}} = y, \mathtt{X}^{\mathtt{H}} = x, \mathtt{A}^{\mathtt{H}} = a, \mathtt{B}^{\mathtt{H}} = b\}\ \mathtt{H}\ \{\mathtt{O}^{\mathtt{H}} = h(y, x; a, b)\}$$

where we have renamed variables such that no variable occur both in G and H. Let $\mathtt{X}, \mathtt{Y}, \mathtt{A}, \mathtt{W}, \mathtt{O}$ be fresh variables and let P be the program

```
X^G:= X; A^G:= A; G; B^H:= O^G; nil(W)
for Y [ Y^H:= W; X^H:= X; A^H:= A; H;
        B^H:= O^H; suc(W) ];
O:= O^H
```

Then the theorem and (*) hold for P. In particular, the relation $\xrightarrow{\mathtt{P}}$ will be irreflexive since both $\mathtt{B}^{\mathtt{H}}$ and $\mathtt{O}^{\mathtt{H}}$ are maximal elements in the ordering $\xrightarrow{\mathtt{H}}$. □

References

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

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

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

[4] 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.

[5] S.J. Bellantoni and K.-H. Niggl. *Ranking primitive recursions: The low Grzegorczyk classes revisited.* SIAM J. of Comput., 29(2):401–415, 2000.

[6] P. Clote. *Computation Models and Function Algebra.* In: Handbook of Computability Theory. Ed Griffor, ed., Elsevier 1996.

[7] P. Clote. *A Safe Recursion Scheme for Exponential time* Technical Report 9607, Computer Science Department, LMU Munich, October 1996.

[8] B. Goetze and W. Nehrlich. *The structure of loop programs and subrecursive hierarchies.* Zeitschr. f. math. Logik und Grundlagen d. Math., 26:255–278, 1980.

[9] A. Grzegorczyk. *Some classes of recursive functions.* Rozprawy Matematyczne, No. IV, Warszawa, 1953.

[10] M. Hofmann. *Typed lambda calculi for polynomial-time computation.* Habilitation Thesis, TU Darmstadt, 1998.

[11] M. Hofmann. (Editor) *Proceedings of the 3rd International Workshop on Implicit Computational Complexity ICC '01.* BRICS Notes Series, Department of Computer Science University of Aarhus. `http://www.brics.dk`, `ftp://ftp.brics.dk`

[12] L. Kristiansen. *Papers on subrecursion theory.* Dr Scient Thesis, ISSN 0806-3036, ISBN 82-7368-130-0, Research Report 217, Department of Informatics, University of Oslo 1996.

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

[14] M. Kutylowski. *Small Grzegorczyk classes.* J. London Math. Soc., 36:193–210, 1987.

[15] D. Leivant. *Stratified functional programs and computational complexity.* In: Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, pp. 325-333, New York, 1993.

[16] D. Leivant and Jean-Yves Marion. *Lambda Calculus Characterizations of Poly-Time.* Fundamenta Informaticae, 19:167–184, 1993.

[17] 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, pp. 320–343, Birkhäuser, 1994.

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

[19] D. Leivant and J.-Y. Marion. *Ramified Recurrence and Computational Complexity II: Substitution and Poly-space.* Computer science logic (Kazimierz, 1994), 486–500, Lecture Notes in Comput. Sci., 933, Springer, Berlin, 1995

[20] D. Leivant and J.-Y. Marion. *Ramified Recurrence and Computational Complexity IV: Predicative Functionals and Poly-space. Information and Computation*, to appear.

[21] D. Leivant. *Intrinsic reasoning about functional programs. Annals of Pure and Applied Logic*, to appear.

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

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

[24] S.S. Muchnick. *The vectorized Grzegorczyk hierarchy.* Z. Math. Logik Grundlag. Math. 22:441–480, 1976.

[25] K.-H. Niggl. *The μ-measure as a tool for classifying computational complexity.* Archive for Mathematical Logic, 39:515–539, 2000.

[26] P. Odifreddi. *Classical recursion theory. Vol. II.* . Studies in Logic and the Foundations of Mathematics, 143. North-Holland Publishing Co., Amsterdam, 1999.

[27] 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.

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

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

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

Recent BRICS Report Series Publications

RS-01-46 **Lars Kristiansen.** ***The Implicit Computational Complexity of Imperative Programming Languages*****. November 2001. 46 pp.**

RS-01-45 **Ivan B. Damgård and Gudmund Skovbjerg Frandsen.** ***An Extended Quadratic Frobenius Primality Test with Average Case Error Estimates*****. November 2001. 43 pp.**

RS-01-44 **M. Oliver Möller, Harald Rueß, and Maria Sorea.** ***Predicate Abstraction for Dense Real-Time Systems*****. November 2001.**

RS-01-43 **Ivan B. Damgård and Jesper Buus Nielsen.** ***From Known-Plaintext Security to Chosen-Plaintext Security*****. November 2001. 18 pp.**

RS-01-42 **Zoltán Ésik and Werner Kuich.** ***Rationally Additive Semirings*****. November 2001. 11 pp.**

RS-01-41 **Ivan B. Damgård and Jesper Buus Nielsen.** ***Perfect Hiding and Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor*****. October 2001. 43 pp.**

RS-01-40 **Daniel Damian and Olivier Danvy.** ***CPS Transformation of Flow Information, Part II: Administrative Reductions*****. October 2001. 9 pp.**

RS-01-39 **Olivier Danvy and Mayer Goldberg.** ***There and Back Again*****. October 2001. 14 pp.**

RS-01-38 **Zoltán Ésik.** ***Free De Morgan Bisemigroups and Bisemilattices*****. October 2001. 13 pp.**

RS-01-37 **Ronald Cramer and Victor Shoup.** ***Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption*****. October 2001. 34 pp.**

RS-01-36 **Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob.** ***Cache Oblivious Search Trees via Binary Trees of Small Height*****. October 2001.**

RS-01-35 **Mayer Goldberg.** ***A General Schema for Constructing One-Point Bases in the Lambda Calculus*****. September 2001. 6 pp.**

RS-01-34 **Flemming Friche Rodler and Rasmus Pagh.** ***Fast Random Access to Wavelet Compressed Volumetric Data Using Hashing*****. August 2001. 31 pp.**