Proceedings of the 3rd International Workshop on

Implicit Computational Complexity ICC '01

Aarhus, Denmark, May 20–21, 2001

Martin Hofmann
(editor)

BRICS Notes Series NS-01-3

ISSN 0909-3206 May 2001

3rd Workshop on

IMPLICIT COMPUTATIONAL COMPLEXITY

Aarhus, 20.5.2001–21.5.2001

Foreword

The synergy between Logic, Computational Complexity and Programming Language Theory has gained importance and vigour in recent years, cutting across areas such as Proof Theory, Computation Theory, Applicative Programming, and Philosophical Logic. Several machine-independent approaches to computational complexity have been developed, which characterize complexity classes by conceptual measures borrowed primarily from mathematical logic. Collectively these approaches might be dubbed IMPLICIT COMPUTATIONAL COMPLEXITY.

Practically, implicit computational complexity provides a framework for a streamlined incorporation of computational complexity into areas such as formal methods in software development, programming language theory.

After previous workshops in Indianapolis (1994), Baltimore (1998), Trento (1999) and Santa Barbara (2000) the present meeting in Aarhus formed another instalment of a series of workshops on these topics. The broad spectrum and the quality of submissions shows that the subject has gained maturity and it is hoped that workshops like this will further contribute to the growth of the field.

The workshop was open to everyone and no formal refereeing took place; the present proceedings are intended mainly as a reference for workshop participants so as to enable better interaction. It is anticipated that participants will submit their work to scholarly fora for publication after the workshop.

Many thanks are due to the local organisers Karen Kjær Møller and Olivier Danvy for all their help; they made my own job a really pleasurable one. I also would like to thank Neil Jones for offering an invited talk (whose abstract is included in these proceedings) and the programme committee consisting of Samson Abramsky, Bruce Kapron, Jean-Yves Marion, Søren Riis, and Helmut Schwichtenberg for advice and informal quality checks of submissions.

Martin Hofmann (programme chair)

Table of contents

Linear Ramified Higher Type Recursion and Parallel Computation (extended abstract)

Klaus Aehlig[1,⋆], Jan Johannsen[2,⋆⋆], Helmut Schwichtenberg[1,⋆⋆⋆], and Sebastiaan A. Terwijn[3,†]

[1] Mathematisches Institut, Ludwig-Maximilians-Universität München, Theresienstraße 39, 80333 München, Germany
{aehlig,schwicht}@rz.mathematik.uni-muenchen.de,
Tel.: +49 89 2394 { -4415, -4413}

[2] Institut für Informatik, Ludwig-Maximilians-Universität München Oettingenstraße 67, 80538 München, Germany
jjohanns@informatik.uni-muenchen.de,
Tel.: +49 89 2178 2209, Fax: +49 89 2178 2238

[3] Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
terwijn@cs.vu.nl, Tel.: +31 20 4447753

Abstract. A typed lambda calculus with recursion in all finite types is defined such that the first order terms exactly characterize the parallel complexity class NC. This is achieved by use of the appropriate forms of recursion (concatenation recursion and logarithmic recursion), a ramified type structure and imposing of a linearity constraint.
Keywords: higher types, recursion, parallel computation, NC, lambda calculus, linear logic, implicit computational complexity

1 Introduction

One of the most prominent complexity classes, other than polynomial time, is the class NC of functions computable in parallel polylogarithmic time with a polynomial amount of hardware. This class has several natural characterizations in terms of circuits, alternating Turing machines or parallel random access machines, as used in this work. It can be argued that NC is the class of efficiently parallalizable problems, just as polynomial time is often viewed as the correct formalization of feasible sequential computation.
Machine-independent characterizations of computational complexity classes are not only of theoretical, but recently also of increasing practical interest. Besides

⋆ Supported by the DFG Graduiertenkolleg “Logik in der Informatik”
⋆⋆ Supported by the DFG Emmy Noether-Programme under grant No. Jo 291/2-1
⋆⋆⋆ The hospitality of the Mittag-Leffler Institute in the spring of 2001 is gratefully acknowledged.
† Supported by a Marie Curie fellowship of the European Union under grant no. ERB-FMBI-CT98-3248

indicating the robustness and naturality of the classes in question, they also provide guidance for the development of programming languages [10].

The earliest such characterizations, starting with Cobham's function algebra for polynomial time [8], used recursions with explicit bounds on the growth of the defined functions. Function algebra characterizations in this style of parallel complexity classes, among them NC, were given by Clote [7] and Allen [1].

More elegant *implicit* characterizations, i.e., without any explicitly given bounds, but instead using logical concepts like ramification or tiering, have been given for many complexity classes, starting with the work of Bellantoni and Cook [4] and Leivant [13] on polynomial time. In his thesis, Bellantoni [2] gives such a characterization of NC using a ramified variant of Clote's recursion schemes. A different implicit characterization of NC, using tree recursion, was given by Leivant [14]. Other parallel complexity classes, viz. parallel logarithmic and polylogarithmic time, were given implicit characterizations by Bellantoni [3], Bloch [6] and Leivant and Marion [15].

In order to apply the approach within the functional programming paradigm, one has to consider functions of higher type, and thus extend the function algebras by a typed lambda calculus. To really make use of this feature, it is desirable to allow the definition of higher type functions by recursion. Higher type recursion was originally considered by Gödel [9] for the analysis of logical systems. Systems with recursion in all finite types characterizing polynomial time were given by Bellantoni et al. [5] and Hofmann [11], based on the first-order system of Bellantoni and Cook [4].

We define an analogous system that characterizes NC while allowing an appropriate form of recursion, viz. logarithmic recursion as used by Bellantoni [2], in all finite types. More precisely, our system is a typed lambda calculus which allows two kinds of function types, denoted $\sigma \multimap \tau$ and $\Box\sigma \multimap \tau$, and two sorts of variables of the ground type ι, the *complete* ones in addition to the usual ones, which are called incomplete for emphasis. A function of type $\Box\sigma \multimap \tau$ can only be applied to complete terms of type σ, i.e., terms containing only complete free variables.

It features two recursion operators LR and CR, the latter corresponding to Clote's [7] concatenation recursion on notation, which can naturally only be applied to first-order functions. The former is a form of recursion of logarithmic length characteristic of all function algebra representations of NC, and here can be applied to functions of all $\Box$-free types. The function being iterated as well as the numerical argument being recurred on have to be complete, i.e., the type of LR is $\sigma \multimap \Box(\Box\iota \multimap \sigma \multimap \sigma) \multimap \Box\iota \multimap \sigma$ for σ $\Box$-free.

The crucial restriction, justifying the use of linear logic notation, is a linearity constraint on variables of higher types: all higher type variables in a term must occur at most once.

The main new contribution in the analysis of the complexity of the system is a strict separation between the term, i.e., the program, and the numerical context, i.e., its input and data. Whereas the runtime may depend polynomially on the former, it may only depend polylogarithmically on the latter.

To make use of this conceptual separation, the algorithm that unfolds recursions computes, given a term and context, a recursion-free term *plus a new context*. In particular, it does not substitute numerical parameters, but only uses them for unfolding; in some cases, including the reduction of CR, it extends the context. This way, the growth of terms in the elimination of recursions is kept under control. In earlier systems that comprised at least polynomial time this strict distinction was not necessary, since the computation time there may depend on the *input* superlinearly. Note that any reasonable form of computation will depend at least linearly on the size of the *program*.

As opposed to the first-order system of Bellantoni [2], the numerical argument that governs a concatenation recursion must be complete in our system, the type of CR is $(\iota \multimap \iota) \multimap \Box\iota \multimap \iota$. The reason is that the amount of hardware required depends exponentially on the number of CR in a term, thus we must not allow duplications of this constant during the unfolding of LR. The only way to avoid this is by the more restrictive typing. This weaker form of concatenation recursion nevertheless suffices to include all of NC, when the set of base functions is slightly extended.

A further feature of this work is the usage of a tree data structure to store numerals during the computation. Whereas trees are used as the principal data structure in other characterizations of parallel complexity classes [14, 15], our system works with usual binary numerals, and trees are only used in the implementation.

2 Formal Definition of the System

We use simple types with two forms of abstraction over a single base type ι, i.e. our types are given by the grammar

$$\sigma, \tau ::= \iota \mid \sigma \multimap \tau \mid \Box\sigma \multimap \tau$$

As the intended semantics for our base type are the binary numerals we have the constants 0 of type ι and s_0 and s_1 of type $\iota \multimap \iota$. Moreover we add the constants half, len of type $\iota \multimap \iota$, bit, drop of type $\iota \multimap \iota \multimap \iota$ and sm of type $\iota \multimap \iota \multimap \iota \multimap \iota$ for the corresponding base functions. We allow case-distinction for arbitrary types, so we have a constant d_σ of type $\iota \multimap \sigma \multimap \sigma \multimap \sigma$ for *every* type σ. Growth is added to the system via the constant $\#$, recursion via the constant LR and parallelism via the constant CR. Their types are

$$\begin{array}{lll l} \# & : & \Box\iota \multimap \iota & \\ \mathsf{CR} & : & (\iota \multimap \iota) \multimap \Box\iota \multimap \iota & \\ \mathsf{LR}_\sigma & : & \sigma \multimap \Box(\Box\iota \multimap \sigma \multimap \sigma) \multimap \Box\iota \multimap \sigma & \sigma\ \Box\text{-free} \end{array}$$

Terms are built from variables and constants via abstraction and typed application. We have incomplete variables of every type, denoted by x, y, ... and complete variables of ground type, denoted by $\mathbf{x}$, $\mathbf{y}$, ... All our variables and terms have a fixed type and we add type superscripts to emphasize the type: x^σ, $\mathbf{x}^\iota$, t^σ. So terms are given by the grammar

$$s, t, \ldots ::= c \mid x^\sigma \mid \mathbf{x}^\iota \mid (\lambda x^\sigma . t^\tau)^{\sigma \multimap \tau} \mid (\lambda \mathbf{x}^\iota . t^\tau)^{\Box\sigma \multimap \tau} \mid (t^{\sigma \multimap \tau}\, s^\sigma)^\tau \mid \left(t^{\Box\sigma \multimap \tau}\, s^\sigma\right)^\tau$$

where in the last case we require s to be complete; a term is called *complete* if all its free variables are. It should be noted that, although we cannot form terms of type $\Box\sigma \multimap \tau$ with $\sigma \neq \iota$ directly via abstraction it is still important to have that type in order to express, for example, that the first argument of LR must not contain free incomplete variables.
In the following we omit the type subscripts at the constants d_σ and LR_σ if the type is obvious or irrelevant. Moreover we identify α-equal terms. As usual application associates to the left. A *binary numeral* is either 0, or of the form $\mathsf{s}_{i_1}(\ldots(\mathsf{s}_{i_k}(\mathsf{s}_1 0)))$. The semantics of ι as binary numerals (rather than binary words) is given by the conversion rule $\mathsf{s}_0\ 0 \mapsto 0$. In the following definitions we identify binary numerals with the natural number they represent. The base functions get their usual semantics, i.e. we add conversion rules $\mathsf{len}\, n \mapsto \lceil \log_2(n+1) \rceil =: ||n||$, $\mathsf{drop}\, n\, m \mapsto \lfloor \frac{n}{2^{||m||}} \rfloor$, $\mathsf{half}\, n \mapsto \lfloor \frac{n}{2^{\lceil ||n||/2 \rceil}} \rfloor$, $\mathsf{bit}\, n\, i \mapsto \lfloor \frac{n}{2^i} \rfloor \bmod 2$, $\mathsf{sm}\, w\, m\, n \mapsto 2^{||m|| \cdot ||n||} \bmod 2^{||w||}$. Moreover, we add the conversion rules

$\mathsf{d}_\sigma\, 0$	$\mapsto$	$\lambda x^\sigma\, y^\sigma\, .\, x$
$\mathsf{d}_\sigma\, (\mathsf{s}_i n)$	$\mapsto$	$\lambda x_0^\sigma\, x_1^\sigma\, .\, x_i$
$\#\, n$	$\mapsto$	$\mathsf{s}_0{}^{\lvert\lvert n \rvert\rvert^2}(\mathsf{s}_1\, 0)$
$\mathsf{CR}\, h\, 0$	$\mapsto$	0
$\mathsf{CR}\, h\, (\mathsf{s}_i\, n)$	$\mapsto$	$\mathsf{d}_{(\iota \multimap \iota)}\ (h\, (\mathsf{s}_i n))\, \mathsf{s}_0\, \mathsf{s}_1\, (\mathsf{CR}\, h\, n)$
$\mathsf{LR}\, g\, h\, 0$	$\mapsto$	g
$\mathsf{LR}\, g\, h\, n$	$\mapsto$	$h\, n\, (\mathsf{LR}\, g\, h\, (\mathsf{half}\, n))$

Here we always assumed that n, m and $\mathsf{s}_i\, n$ are binary numerals, and in particular that the latter does not reduce to 0. In the last rule, n has to be a binary numeral different from 0.
As usual the reduction relation is the closure of $\mapsto$ under all term forming operations and equivalence is the symmetric, reflexive, transitive hull of the reduction relation. As all reduction rules are correct with respect to the intended semantics and obviously all closed normal terms of type ι are numerals, closed terms t of type ι have a unique normal form that we denote by t^{nf}.
As usual, lists of notations for terms/numbers/ ... that only differ in successive indices are denoted by leaving out the indices and putting an arrow over the notation. It is usually obvious where to add the missing indices. If not we add a dot wherever an index is left out. Lists are inserted into formulae "in the natural way", e.g., $\overrightarrow{hm.} = hm_1, \ldots, hm_k$ and $x\overrightarrow{t} = ((x\, t_1) \ldots t_k)$ and $|g| + \overrightarrow{|s|} = |g| + |s_1| + \ldots + |s_k|$.
As already mentioned, we are not interested in all terms of the system, but only in those fulfilling a certain linearity condition.

Definition 1. *A term t is called* linear, *if every variable of higher type in t occurs at most once.*

3 Soundness

Definition 2. *The length $|t|$ of a term t is inductively defined as follows: For a variable x, $|x| = 1$, and for any constant c other than d, $|c| = 1$, whereas $|\mathsf{d}| = 3$. For complex terms we have the usual clauses $|r\,s| = |r| + |s|$ and $|\lambda x.r| = |r| + 1$.*

The length of the constant d is motivated by the desire to decrease the length of a term in the reduction of a d-redex.

Definition 3. *For a list $\vec{n}$ of numerals, define $|\vec{n}| := \max(\overrightarrow{|n|})$.*

Definition 4. *A context is a list of pairs (x, n) of variables of type ι and numerals, where all the variables are distinct. If $\vec{x}$ is a list of distinct variables of type ι and $\vec{n}$ a list of numerals of the same length, then we denote by $\vec{x}; \vec{n}$ the context $\overrightarrow{(x, n)}$.*

Definition 5. *For every symbol c of our language and term t, $\sharp_c(t)$ denotes the number of occurrences of c in t. For obvious esthetic reasons we abbreviate $\sharp_{\#}(t)$ by $\sharp(t)$.*

Definition 6. *A term t is called* simple *if t contains none of the constants $\#$, CR or LR.*

By a simple induction on $|t|$ we can show an upper bound for the length of numerals:

Lemma 1. *Let t be a simple, linear term of type ι and $\vec{x}; \vec{n}$ a context, such that all free variables in t are among $\vec{x}$. Then for $t^* := t[\vec{x} := \vec{n}]^{\mathrm{nf}}$ we have $|t^*| \leq |t| + |\vec{n}|$.*

Data Structure

We represent terms as parse trees, fulfilling the obvious typing constraints. The number of edges leaving a particular node is called the out-degree of this node. There is a distinguished node with in-degree 0, called the root. Each node is stored in a record consisting of an entry `cont` indicating its kind, plus some pointers to its children. We allow the following kinds of nodes with the given restrictions:

- Variable nodes representing a variable x. Every variable has a unique name and an associated register $\mathtt{R}[x]$.
- Abstraction nodes λx representing the binding of the variable x.
- For each constant c, there are nodes representing the constant c.
- Application nodes @ representing the application of two terms. The obvious typing constraints have to be fulfilled.

- Auxiliary nodes κ_i representing the composition of type one. These nodes are labeled with a natural number i, and each of those nodes has out-degree either 2 or 3. They will be used to form 2/3-trees (as e.g. described by Knuth [12]) representing numerals during the computation. We require that any node reachable from a $\kappa_\cdot$-node is either a $\kappa_\cdot$ node as well or one of the constants s_0 or s_1.
- Auxiliary nodes κ' representing the identification of type-one-terms with numerals (via "applying" them to 0). The out-degree of such a node, which is also called a "numeral node", either is zero, in which case the node represents the term 0, or the out-degree is one and the edge starting from this node either points to one of the constants s_0 or s_1 or to a $\kappa_\cdot$ node.
- Finally, there are so-called dummy nodes $\diamond$ of out-degree 1. Dummy nodes serve to pass on pointers: a node that becomes superfluous during reduction is made into a dummy node, and any pointer to it will be regarded as if it pointed to its child.

A tree is called a *numeral* if the root is a numeral node, all leaves have the same distance to the root and the label i of every κ_i node is the number of leaves reachable from that node. By standard operations on 2/3-trees it is possible in sequential logarithmic time to

- split a numeral at a given position i.
- find out the i'th bit of the numeral.
- concatenate two numerals.

So using κ' and $\kappa_\cdot$ nodes is just a way of implementing "nodes" labeled with a numeral allowing all the standard operations on numerals in logarithmic time. Note that the length of the label i (coded in binary) of a κ_i node is bounded by the logarithm of the number of nodes.

Normalization Algorithms and Their Complexity

Lemma 2. *Let t be a simple, linear term of type ι and $\vec{x}; \vec{n}$ a context such that all free variables in t are among the $\vec{x}$. Then the normal form of $t[\vec{x} := \vec{n}]$ can be computed in time $O(|t| \cdot \log |\vec{n}|)$ by $O(|t| \cdot |\vec{n}|)$ processors.*

Proof. Explicitly define and analyze an algorithm noting that that linear beta-redexes can be reduced via alternating pointers and all the relevant copying operations and computation of the base functions can be performed on our 2/3-tree representation of numerals in time $O(\log |\vec{n}|)$ by $O(|\vec{n}|)$ processors. $\square$

Let $f(n) \lesssim g(n)$ abbreviate $f(n) \leq (1 + o(1))g(n)$, i.e., $\limsup_{n\to\infty} \frac{f(n)}{g(n)} \leq 1$.

Lemma 3. *Let t be a linear term of $\square$-free type and $\vec{x}; \vec{n}$ a context with all free variables of $t[\vec{x} := \vec{n}]$ incomplete. Then there are a term* $\mathrm{simp}(t, \vec{x}; \vec{n})$ *and a context $\vec{y}; \vec{m}$ such that* $\mathrm{simp}(t, \vec{x}; \vec{n})[\vec{y} := \vec{m}]$ *is simple and equivalent to*

$t[\vec{x} := \vec{n}]$, *and which can be computed in time* $\lesssim 2^{\sharp_{\mathsf{LR}}(t)} \cdot |t| \cdot (2^{\sharp(t)} \cdot \log |\vec{n}|)^{\sharp_{\mathsf{LR}}(t)+2}$ *by* $\lesssim |t| \cdot |\vec{n}|^{2^{\sharp(t)}(\sharp_{\mathsf{CR}}(t)+\sharp_{\mathsf{LR}}(t)+2)}$ *processors, such that*

$$|\mathrm{simp}(t, \vec{x}; \vec{n})| \lesssim |t| \cdot \left(2^{\sharp(t)} \cdot \log |\vec{n}|\right)^{\sharp_{\mathsf{LR}}(t)} \quad \textit{and} \quad |\vec{m}| \lesssim |\vec{n}|^{2^{\sharp(t)}} \; .$$

Proof. By induction on $\sharp_{\mathsf{LR}}(t)$, with a side-induction on $|t|$ show that the following algorithm does it:
By pattern matching, determine in time $O(|t|)$ the form of t, and branch according to the form.

- If t is a variable or one of the constants 0 or d, then return t and leave $\vec{x}; \vec{n}$ unchanged.
- If t is $c\,\vec{s}$, where c is one of the constants s_i, drop, bit, len or sm then recursively simplify $\vec{s}$, giving $\vec{s^*}$ and contexts $\vec{y_j}; \vec{m_j}$, and return $c\,\vec{s^*}$ and $\vec{\vec{y}}; \vec{\vec{m}}$.
- If t is $\mathsf{d}\,r\,\vec{s}$, then simplify r giving r' and $\vec{y}; \vec{m}$. Compute the numeral $r^* := r'[\vec{y} := \vec{m}]^{\mathrm{nf}}$, and reduce the redex $\mathsf{d}\,r^*$, giving t', and recursively simplify $t'\,\vec{s}$ with context $\vec{x}; \vec{n}$.
- If t is $\#\,r$ then simplify r giving r' and $\vec{y}; \vec{m}$. Compute the numeral $r^* := r'[\vec{y} := \vec{m}]^{\mathrm{nf}}$, and return a new variable y' and the context $y'; 2^{|r^*|^2}$.
- If t is $\mathsf{CR}\,h\,r$, then simplify r giving r' and $\vec{y}; \vec{m}$, and compute the numeral $r^* := r'[\vec{y} := \vec{m}]^{\mathrm{nf}}$.
  Spawn $|r^*|$ many processors, one for each leaf of r^*, by moving along the tree structure of r^*. The processor at bit i of r^* simplifies $h\,z$ in the context $\vec{x}, z; \vec{n}, \lfloor r^*/2^i \rfloor$, giving a term h_i and context $\vec{y_i}; \vec{m_i}$, then he computes $h_i^* := h_i[y_i := m_i]^{\mathrm{nf}}$, retaining only the lowest order bit b_i.
  The bits $\vec{b}$ are collected into a 2/3-tree representation of a numeral m, which is output in the form of a new variable z and the context $z; m$.
- t is $\mathsf{LR}\,g\,h\,m\,\vec{s}$ then simplify m, giving m' and $\vec{x_m}; \vec{n_m}$. Normalize m' in the context $\vec{x}, \vec{x_m}; \vec{n}, \vec{n_m}$, giving m^*. Form k numerals $m_i = \mathsf{Half}^i(m^*)$ and sequentially simplify $\vec{h\,m_i}$, giving $\vec{h'}$. (Of course, more precisely simplify $h\,x$ for a new variable x in the context extended by $x; m_i$.) Then form the term

$$t' := h_0'(h_1' \ldots (h_k'\,g))\,\vec{s}$$

  and simplify it.
- If t is of the form $\lambda x.r$ then recursively simplify r.
- If t is of the form $(\lambda x.r)\,s\,\vec{s}$ and x occurs at most once in r then recursively simplify $r[x := s]\,\vec{s}$.
- If t is of the form $(\lambda x.r)\,s\,\vec{s}$ and x occurs several times in r, then simplify s giving s' and a context $\vec{y}; \vec{m}$. Normalize s' in this context giving the numeral s^*. Then simplify $r\,\vec{s}$ in the context $\vec{x}, x; \vec{n}, s^*$. $\square$

Theorem 1. *Let t be a linear term of type $\vec{\Box\iota} \multimap \iota$. Then the function denoted by t is in NC.*

Proof. Let $\overrightarrow{n}$ be an input, given as 2/3-tree representations of numerals, and $\overrightarrow{\mathbf{x}}$ complete variables of type ι. Using Lemma 3, we compute $t' := \mathrm{simp}(t\,\overrightarrow{\mathbf{x}}, \overrightarrow{\mathbf{x}}; \overrightarrow{n})$ and a new context $\overrightarrow{y}; \overrightarrow{m}$ with $|t'| \leq (log\,|\overrightarrow{n}|)^{O(1)}$ and $|\overrightarrow{m}| \leq |\overrightarrow{n}|^{O(1)}$ in time $(\log|\overrightarrow{n}|)^{O(1)}$ by $|\overrightarrow{n}|^{O(1)}$ many processors.
Then using Lemma 2 we compute the normal form $t'[\overrightarrow{y} := \overrightarrow{m}]^{\mathrm{nf}}$ in time $O(|t'| \cdot \log|\overrightarrow{m}|) = (\log|\overrightarrow{n}|)^{O(1)}$ by $O(|t'|\,|\overrightarrow{m}|) = |\overrightarrow{n}|^{O(1)}$ many processors.
Hence the function denoted by t is computable in polylogarithmic time by polynomially many processors, and thus is in NC. □

4 Completeness

Bellantoni [2] defines a two-sorted function algebra 2CLO characterizing NC, which is an implicit variant of Clote's function algebra **A** [7] that characterizes NC by concatenation recursion and logarithmic recursion with explicit bounds. 2CLO is a class of functions of two sorts of arguments, the normal inputs written to the left of a separating semicolon, and the safe inputs written to the right. It is defined to be the smallest class of functions that contains the constant zero, projections $\pi_j^{m,n}(x_1, \ldots, x_m; x_{m+1}, \ldots, x_{m+n}) = x_j$, successors S_i, conditional D, bit test Bit, binary length Len, smash $\#'(w; a, b) = 2^{||a|| \cdot ||b||} \bmod 2^{||w||^2}$ and is closed under the following operations:

- *Safe composition*: from g, h and k define $f(\overrightarrow{x}; \overrightarrow{y}) := g(\overrightarrow{h}(\overrightarrow{x};)\,;\,\overrightarrow{k}(\overrightarrow{x}; \overrightarrow{y}))$.
- *Concatenation recursion*: from h define f by

$$\begin{aligned} f(\overrightarrow{x}; 0, \overrightarrow{a}) &= 0 \\ f(\overrightarrow{x}; \mathsf{S}_i(;b), \overrightarrow{a}) &= \mathsf{S}_{h(\overrightarrow{x};b,\overrightarrow{a}) \bmod 2}(; f(\overrightarrow{x}; b, a)) \qquad \text{for } \mathsf{S}_i(;b) > 0 \end{aligned}$$

- *Log recursion*: from g and h define f by

$$\begin{aligned} f(0, \overrightarrow{x}; \overrightarrow{a}) &= g(\overrightarrow{x}; \overrightarrow{a}) \\ f(y, \overrightarrow{x}; \overrightarrow{a}) &= h(y, \overrightarrow{x}; \overrightarrow{a}, f(\mathsf{Half}(;y), \overrightarrow{x}; \overrightarrow{a})) \qquad \text{for } y > 0 \end{aligned}$$

It is proved in Chapter 7 of Bellantoni's thesis [2] that this function algebra characterizes NC:

Theorem 2 (Bellantoni [2]). *A function $f(\overrightarrow{x})$ is in NC if and only if $f(\overrightarrow{x};) \in$ 2CLO.*

We now define a modified version of the class 2CLO, denoted by 2NC, as the smallest class that contains all the base functions of 2CLO and additionally the functions Half and Drop with $\mathsf{Half}(;a) = \lfloor a/2^{\lceil ||a||/2 \rceil} \rfloor$ and $\mathsf{Drop}(;a,b) = \lfloor a/2^{||b||} \rfloor$, and is closed under safe composition and log recursion and the following variant of concatenation recursion:

$$\begin{aligned} f(0, \overrightarrow{x}; \overrightarrow{a}) &= 0 \\ f(\mathsf{S}_i(;y), \overrightarrow{x}; \overrightarrow{a}) &= \mathsf{S}_{h(y,\overrightarrow{x};\overrightarrow{a}) \bmod 2}(; f(y, \overrightarrow{x}; \overrightarrow{a})) \qquad \text{for } \mathsf{S}_i(;y) > 0\ . \end{aligned}$$

where the input that governs the recursion has to be normal.

Lemma 4. *A function $f(\overrightarrow{x})$ is in NC if and only if $f(\overrightarrow{x};) \in$ 2NC.*

Proof. We show how to modify the proof of the corresponding proposition for 2CLO in [2]. The proof of the "if" direction can be left unchanged, one only has to observe that the functions Half and Drop are in NC, and that the Bounding Lemma still holds.
For the "only if" part, one has to show that for every function $f(\overrightarrow{x})$ in NC there is a function $f'(w;\overrightarrow{x})$ in 2NC and a polynomial p_f such that $f'(w,\overrightarrow{x}) = f(\overrightarrow{x})$ for all w with $||w|| \geq p_f(\overrightarrow{||x||})$. This is proved by induction on the definition of f in Clote's function algebra **A**. $\square$

By embedding 2NC in the obvious way we can prove that our term system can denote all functions in NC.

Theorem 3. *For every function $f(\overrightarrow{x};\overrightarrow{y})$ in 2NC, there is a closed linear term t_f of type $\square\overrightarrow{\iota} \multimap \overrightarrow{\iota} \multimap \iota$ that denotes f.*

From Theorems 3 and 1 we immediately get our main result:

Corollary 1. *A number-theoretic function f is in NC if and only if it is denoted by a linear term of our system.*

References

1. B. Allen. Arithmetizing uniform NC. *Annals of Pure and Applied Logic*, 53(1):1–50, 1991.
2. S. Bellantoni. *Predicative Recursion and Computational Complexity.* PhD thesis, University of Toronto, 1992.
3. S. Bellantoni. Characterizing parallel time by type 2 recursions with polynomial output length. In D. Leivant, editor, *Logic and Computational Complexity*, pages 253–268. Springer LNCS 960, 1995.
4. S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.
5. S. Bellantoni, K.-H. Niggl, and H. Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104:17–30, 2000.
6. S. Bloch. Function-algebraic characterizations of log and polylog parallel time. *Computational Complexity*, 4:175–205, 1994.
7. P. Clote. Sequential, machine independent characterizations of the parallel complexity classes *ALogTIME*, AC^k, NC^k and *NC*. In S. Buss and P. Scott, editors, *Feasible Mathematics*, pages 49–69. Birkhäuser, 1990.
8. A. Cobham. The intrinsic computational difficulty of functions. In *Proceedings of the second International Congress on Logic, Methodology and Philosophy of Science*, pages 24–30, 1965.
9. K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.
10. M. Hofmann. Programming languages capturing complexity classes. *ACM SIGACT News*, 31(2), 2000. Logic Column 9.
11. M. Hofmann. Safe recursion with higher types and BCK-algebra. *Annals of Pure and Applied Logic*, 104:113–166, 2000.

12. D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 2nd edition, 1998.
13. D. Leivant. Stratified functional programs and computational complexity. In *Proc. of the 20th Symposium on Principles of Programming Languages*, pages 325–333, 1993.
14. D. Leivant. A characterization of NC by tree recurrence. In *Proc. 39th Symposium on Foundations of Computer Science*, pages 716–724, 1998.
15. D. Leivant and J.-Y. Marion. A characterization of alternating log time by ramified recurrence. *Theoretical Computer Science*, 236:193–208, 2000.

Feasibility, Provability and Restrictions to von Neumann Architecture

Salvatore Caporaso, Vito L. Plantamura

Dipartimento di Informatica dell'Università di Bari
(caporaso|plantamura)@di.uniba.it

1 Introduction

We wish to present here a transfinite hierarchy $\mathcal{PAL}_\alpha$ ($\alpha < \epsilon_0$):
(a) based, at limits, on a constructive and unlimited operator;
(b) using, at successors, a form of predicative or safe recursion;
(c) fast enough to capture the class $\mathcal{E}^*$ of all functions provably total in Peano Arithmetic;
(d) slow enough to capture classes like DTIMEF$(f(n))$ for $f(n) = n^c$ or $f(n) = n_c(n)$;
(e) in a way consistent with the spirit of *Implicit Computational Complexity* (ICC).
To this purpose, let us consider a computing device R_3 with a read-only *input* register I, a *scratch* register S, and a *program* register P. Unlike von Neumann architecture, no flow of information between S and P is allowed. P decides what has to be carried-out on S, while changes in P are determined by the, so to say, *hardware* of R_3.
Let $\mathcal{B}$ denote the class of all recursion-free Lisp programs in which either argument of each constructor is a constant. The programs for R_3 are the closure of $\mathcal{B}$ under iteration for $|I|$ times. Let $\mathcal{PAL}_{c+1}$ denote the programs defined by a single iteration of a program in $\mathcal{PAL}_{\leq c}$. We have DTIMEF$(n^c) \subseteq \mathcal{PAL}_c \subseteq$ DTIMEF(n^{c+1}). Hence $\mathcal{PL} := \mathcal{PAL}_{<\omega}$ is a syntactic resource-free characterization of poly-time.
If, under certain conditions, R_3 is allowed to copy some information from S into P, we obtain a hierarchy $\mathcal{PAL}_\alpha$ such that $\mathcal{PAL}_{<\epsilon_0} = \mathcal{E}^*$. Its elements are programs of the form

$$g \text{ ite} \quad (g \in \mathcal{PAL}_{\leq\beta}) \quad \text{if } \alpha = \beta + 1; \qquad\qquad g\,\lambda \quad (g \in \mathcal{PL}) \quad \text{if } \alpha \text{ is the limit } \lambda.$$

Assume that all data y have been associated with an ordinal $o[y] < \epsilon_0$, and that codes f^*, α^* have been assigned to the programs f and the ordinals α. The behaviour of R_3 when an input x is stored in both I and S, and when the code for a program is in P, is described by

```
whileP not empty begin
  $X := head[P]$;
  if   $X = g^*$ and $g \in \mathcal{B}$ then $S := g[S]$;
  if   $X = (g \text{ ite})^*$ then $P := (g^*, \dots, g^*, tail[P])$;          ($|I|$ copies of $g^*$)
  if   $head[P] = \lambda^*$ and $o[S] < \lambda$ then begin $P := (tail[S], tail[P])$; $S := head[S]$ end
  end
end
 if $P$ is empty then return $S$ else return ill-syntax.
```

In other words, for all $f \in \mathcal{PAL}_\lambda$ we have ($\{x\}$ is the program coded by x)

$$f[x] = \{g[x]\}[x] \qquad \text{for some } g \in \mathcal{PL} \text{ such that } \{g[x]\} \in \mathcal{PAL}_{<\lambda}. \tag{1}$$

Subrecursive hierarchies $\mathcal{Z}_\alpha$ are usually defined in two steps, both disagreeing with the ICC philosophy. A transfinite sequence of *hierarchy generating functions* Z_α is introduced first in an impredicative and/or unconstructive way. Classes $\mathcal{Z}_\alpha$ are then defined by closure of $\mathcal{Z}_{<\alpha} + Z_\alpha$ under *limited* PR and substitution. At limits, Z_λ is obtained by a *diagonalization scheme* $Z_\lambda(n) := Z_{\lambda[n]}(n)$, which is an act of mere definition; in the sense that the values $Z_{\lambda[n]}(n)$ are not assumed to be generated uniformly in n, and, therefore, existence of an algorithm for Z_λ is not implied. At successors, there are four possibilities: *outer* vs. *inner* recursion; and safe vs. full iteration. According to the former distinction we get respectively

$$O_{\alpha+1}(n) := g(O_\alpha(n)); \qquad I_{\alpha+1}(n) := I_\alpha(g(n));$$

by *safe iteration* we obtain $B_{\alpha+1}(n) := B^n_\alpha(n)$, while full iteration gives all variants of the *fast* (extended Grzegorczyk) hierarchy $E_{\alpha+1}(n) := E^n_\alpha(n)$. Let us say that an algebra is *local* if, unlike λ-calculus or full Lisp, it doesn't access to its whole arguments. If we take as Z_0 a local function like the successor or a program in $\mathcal{PAL}_0$, O and I give the *slow-growing* and the *Hardy* hierarchies G_α and H_α. The latter is closer to the fast-hierarchy than to the slow one, since G_{ϵ_0} grows like H_{ω^3}, while E_α grows like H_{ω^α}. By over-rating the influence of adopting, in the outer definition of $B_{\alpha+1}$, a sequence of step-functions B_α monotonic in α, one might guess that B_α, if not fast-growing like H_α, is at least much faster than G_α. So it is not: we have $B_\alpha(n) = G_{\omega^\alpha}(n)$; hence G_α is as close to B_α as H_α is close to E_α.

In terms of ICC, one would prefer the outer hierarchies, which are *point-wise*, in the sense that the value of O_α at n depends on the value at n of some previous O_β, not on values at points $m \neq n$. However $\mathcal{E}^*$ is too large for them. Indeed, we have $\mathcal{H}_{<\epsilon_0} = \mathcal{E}_{<\epsilon_0} = \mathcal{E}^*$, while: (1) an ordinal lying midway between ϵ_0 and Fefermann's Γ_0 is needed for a hierarchy covering $\mathcal{E}_{<\omega}$ (= PR functions, see [1]); (2) the whole Γ_0 is needed for $\mathcal{E}_\omega$; (3) we get $\mathcal{G}_{<H} = \mathcal{E}^*$ at the not-countable *Howard ordinal* H (see remark above on unconstructivity of Z_λ).

To introduce our hierarchies (in this paper and elsewhere), we reverse the order of the two steps above. $\mathcal{PAL}_\alpha$ is defined first by means of unlimited and constructive operators; the hierarchy-generating sequences come afterwards, just to have an *a posteriori* estimate of complexity and size. In order to achieve our initial goal (d), by fulfilling at the same time conditions (b) and (e), at limits we need a pointwise operator; but, in this case, demand (a) and promise (c) appear to be incompatible. A way to speed-up the functions growth at limits, while keeping them safe at successors is obtained by clause "$\{g[x]\} \in \mathcal{Z}_\alpha$ for some $\alpha < \lambda$" in equation (1) above. It replaces the stronger condition $\{g[x]\} \in \mathcal{Z}_{\lambda[x]}$ previously used in Caporaso *et al.* [1, 2001] (by analogy with $Z_\lambda(n) = Z_{\lambda[n]}(n)$).

We owe to Weiermann [4, 1999] the idea that one can speed-up an outer hierarchy by altering the diagonalization scheme. We look forward to the workshop for an opportunity of comparing our machinery with the beautiful *intrinsic* characterizations of Poly-time, $\mathcal{E}_3$ and $\mathcal{E}^*$ in Leivant [2, 1995] and [3, 2001].

2 Another characterization of Polytime

Note 1 Lisp data consist of *S-expressions*, i.e. words $u, \ldots, z$ of the form $(y \cdot z)$, where y and z are S-expressions or *atoms* $a, b, a_1, \ldots$ (symbols from a given finite alphabet **AT**, denoted by sequences of capital letters). All S-expressions of the form $(x_n \cdot (\ldots (x_1 \cdot \mathrm{NIL}) \ldots))$ for $n \geq 0$ are called *lists*, and denoted by $(x_n, \ldots, x_1)$. For $n = 0$ we have the *empty list* ( ) which equals the atom NIL . We write A^c for the list $(A, \ldots, A)$ (c times). In particular, $\underline{m} := Z^m$ is the *unary numeral* for m. The atom T is used for the truth-value *true.* The *length* $|x|$ of x is the number of its atoms.

We now define a class of programs for the abstract machine R_3 outlined in the Introduction.

Definition 2 The *basic programs* are the following de/con-structors and test

$$\begin{array}{rclrclrcl} \mathrm{ca}[a] = \mathrm{cd}[a] & := & a; & \mathrm{ca}[(x \cdot y)] & := & x; & \mathrm{cd}[(x \cdot y)] & := & y \\ \mathrm{cons^L}_y[x] & := & (y \cdot x); & \mathrm{cons^R}_y[x] & := & (x \cdot y); & \mathrm{eq}_a[x] & := & T \quad \text{iff } x = a. \end{array}$$

$\mathcal{PAL}_0$ is the closure of the basic programs under the constructs

$$g\,h\,\mathrm{sbst} \qquad e\,g\,h\,\mathrm{cond} \qquad e, g, h \in \mathcal{PAL}_0.$$

Assume defined all $\mathcal{PAL}_{\leq\alpha}$. $\mathcal{PAL}_{\alpha+1}$ is the class of all programs of the form

$$h\,g\,\mathrm{ite}\,\mathrm{sbst} \qquad (h \in \mathcal{PAL}_0, \text{ or absent together with sbst; } g \in \mathcal{PAL}_{\leq\alpha}).$$

The class of all *Poly-time Lisp* programs is given by $\mathcal{PL} := \mathcal{PAL}_{<\omega}$.

Notation 3 By expressions of the form $x\;y\;z \overset{\cdot}{\Rightarrow} u\;w$ we mean that a number n of applications of the *productions* below take the contents x, y, z of the registers I, S, P of R_3 into x, u, w. For $n = 1$, we omit the dot.

Definition 4 The interpretation of $\mathcal{PL}$ is given by (z possibly absent, together with the parentheses, on both sides)

1	x	y	(f^*, z)	$\Rightarrow$	$f[y]$	z	f is a basic program
2	x	y	$(g\,h\,\mathrm{sbst}^*, z)$		y	(g^*, h^*, z)	
3.1	x	y	$(e\,g\,h\,\mathrm{cond}^*, z)$		y	(g^*, z)	*if* $x\;y\;(e^*, z) \overset{\cdot}{\Rightarrow} T\;z$
3.2	x	y	$(e\,g\,h\,\mathrm{cond}^*, z)$		y	(h^*, z)	otherwise
4	x	y	$(g\,\mathrm{ite}^*, z)$		y	$(g^*, \ldots, g^*, z)$	$\lvert x \rvert$ copies of g^*.

Notation 5 In principle, $f[x] = y$ means that for all z we have $x, x,\ (f^*, z) \overset{\cdot}{\Rightarrow} x, y, z$. In practice, we let the context decide whether $f[x]$ denotes the output of program f by input x, or the function computed by the program f.
To improve readability we often write $g[h[x]]$ instead of $g\,h\,\mathrm{sbst}[x]$.
f^c is the program built-up from f by $c - 1$ sbst's, and $f\,\mathrm{ite}^c$ is the one built-up by c ite's.

3 Another characterization of the provable functions

Notation 6 $\alpha, \beta, \gamma, \delta, \lambda, \mu, \nu$ will denote ordinals below ϵ_0; in particular, λ, μ, ν are limits. We write $\alpha =_{NF} \beta + \gamma$ (we write $\alpha =_{nf} \gamma + \beta$) when β is the highest (lowest) term of the Cantor form for α, and $\gamma \geq 0$ is the sum of the other terms.
Given a function $G(X)$, define $G \uparrow 0[X] := X; G \uparrow n+1[X] := G[G \uparrow n[X]]$. We write $G \uparrow_{\geq} m[X] = Y$ when we have $G \uparrow n[X] = Y$ for some $n \geq m$.

Definition 7 $\mathcal{PAL}_\lambda$ is the class obtained by adding to $\mathcal{PAL}_{\alpha<\lambda}$ all programs of the form

$$h\ g\ \lambda\,\mathrm{sbst} \qquad (h \in \mathcal{PAL}_0 \text{ or absent; } g \in \mathcal{PL}).$$

Define the class of all *PA Lisp* programs by $\mathcal{PAL} := \mathcal{PAL}_{\alpha<\epsilon_0}$.

Definition 8 1. The *codes* f^* and α^* for the program f and the ordinal α are given by

$\mathrm{ca}^* := (CA);$	$\mathrm{cd}^* := (CD);$	$\mathrm{eq}_y^* := (Q, y, EQ);$
${\mathrm{cons}^{\mathrm{L/R}}}_y{}^* := (Q, y, COL/R);$	$g\ h\,\mathrm{sbst}^* := (g^*, h^*, SB);$	$(g\,\mathrm{ite})^* := (g^*, IT);$
$e\ g\ h\,\mathrm{cond} := (e^*, g^*, h^*, CND);$	$(g\ \lambda)^* := (g^*, \lambda^*)$	
$0^* := \mathrm{NIL}\ ;$	$1^* := U;$	
$(\omega^\alpha)^* := ((), \alpha^*)$ for $\alpha > 0;$	$\alpha^* := (\beta^*, \gamma^*)$	for $\alpha =_{NF} \beta + \gamma$.

Exceptions. $f^{c\,*}$ is (f^*, SB^{c-1}) and $f\ \mathrm{ite}^{c\,*}$ is (f^*, IT^c) (cf. Note 1 for A^c).
Define $|f| := |f^*|$ and $|\alpha| := |\alpha^*|$.
2. Assign an ordinal $o[x] < \epsilon_0$ to every x by

$o[\mathrm{NIL}]$	$:=$	$0;$	
$o[x]$	$:=$	1	x is not a list, or its form is $(Q, y);$
$o[(x_1, \ldots, x_n)]$	$:=$	$o[x_1] + \ldots + o[x_n]$	$x_i \neq \mathrm{NIL}$ for all $i;$
$o[(x_1, \ldots, x_n)]$	$:=$	$\omega^{o[x_1]+\ldots+o[x_n]}$	otherwise.

Note 9 We have $o[\alpha^*] = \alpha$ and $o[f] < \omega$ for all $f \in \mathcal{PL}$. Hence $o[(f\ \lambda)^*] = \lambda$.

Definition 10 $\theta[x]$ is the result of dropping the leftmost NIL (if any) occurring in x. $\theta[x; \lambda]$ is $\theta \uparrow i[x]$ for $i := (\min j \geq 0)(\theta \uparrow j[x] < \lambda)$.

Transfer of information from S into P is associated with the $\mathcal{PAL}$-programs.

Definition 11 The interpretation of $\mathcal{PAL}$ is given by (z and parentheses like in Def. 4)

5	x	y	$((g\ \lambda)^*, z)$	$\Rightarrow$	y	(g^*, λ^*, z)
6	x	(u, w)	(λ^*, z)		u	$(\theta[w, \lambda], z).$

Note 12 1. The *standard assignment of fundamental sequences* $\lambda[n]$, the *Hardy* and the *Grzegorczyk extended* hierarchy-generating functions $H_\alpha(n)$ and $E_\alpha(n)$ are defined by

$$\begin{array}{llll}
\omega^{\alpha+1}[n] := \omega^\alpha n; & \omega^\mu[n] := \omega^{\mu[n]}; & (\mu+\omega^\alpha)[n] := \mu + \omega^\alpha[n] & \text{if } \mu \geq \omega^\alpha; \\
H_0(n) := n; & H_{\alpha+1}(n) := H_\alpha(n+1); & H_\lambda(n) := H_{\lambda[n]}(n); & \\
E_1(n) := n^2+2; & E_{\alpha+1}(n+1) := E_\alpha(E_{\alpha+1}(n)); & E_\lambda(n) := E_{\lambda[n]}(n) & E_{\alpha+1}(0) := 2.
\end{array}$$

2. $\mathcal{E}_\alpha$ $(\omega \leq \alpha)$ is the closure of $\mathcal{E}_{\beta<\alpha}$ under substitution and limited PR.

Theorem 13 1. $\text{DTIMEF}(n^c) \subseteq \mathcal{PAL}_c \subseteq \text{DTIMEF}(n^{c+1})$ $\quad(0 < c < \omega)$.
2. $\text{DTIMEF}(H_\alpha(n)) \subseteq \mathcal{PAL}_\alpha \subseteq \mathcal{E}_\alpha$ $\quad(3 \leq \alpha < \epsilon_0)$.
Hence a function $F(x)$ is computable in polynomial time (is provably total in PA) iff it is computed by a program belonging to $\mathcal{PL}$ (belonging to $\mathcal{PAL}$).

Proof. The left inclusions by Lemmas 17 and 18; the right ones by Lemma 21.

Note 14 The spread under part 1 can be eliminated if we replace $\mathcal{PAL}_0$ with another algebra (see [1]); the one under part 2 can be reduced by means of a more accurate analysis.

4 Simulation of TM's

Notation 15 1. We'll reatrict ourselves to many-tapes TM's M over the alphabet $\mathbf{AT}_+ := \mathbf{AT} \bigcup \{\cdot, (,)\}$. $M[x_1; \ldots; x_n] = z\ [\tau]$ means that M, by *input* the x's on tapes $1, \ldots, n$, writes its *output* z in tape $n+1$ within τ steps, and enters an endless loop over last symbol of z. Sometimes we omit the part $[\tau]$.
2. Given a coding M^* of the TM's into the lists, M_u is the TM coded by u (that is, $M_u^* = u$).

Note 16 The following assertions are proved in [1]:
1. If runtime for $M[x; y]$ is in $O(n^c)$ $(c > 0)$, then there is M_u whose runtime is in $O(n^c)$ too, and such that $M_u[y] = M[u; y]$ (a variant of the recursion theorem).
2. A coding of the instantaneous descriptions (ID) of any given TM M into the lists can be defined, together with a function $nx_M[x] \in \mathcal{PAL}_0$ taking such ID's into the next ones. Moreover, a quadratic time TM NX can be defined, which returns the code for function nx_M uniformly in M^*, in the sense that for all x we have $\text{NX}[x] = nx_{M_x}{}^*$.
3. A quadratic time TM FS can be defined such that $\text{FS}[(x, \lambda^*)] = (x, \lambda[2|x|]^*)$.

Lemma 17 1. For all $f \in \mathcal{PAL}_0$ we have $f\ \text{ite}^c[x] = f \uparrow n^c[x]$ $\quad(n := |x|)$.
2. $M[x] = y\ [kn^c]$ implies $nx_M^k\ \text{ite}^c[x] = y$ (see Notat. 5 for f^k and ite^c).
3. Hence we have $\text{DTIMEF}(n^c) \subseteq \mathcal{PAL}_c$.

Proof. 1. Induction on c, using prod. 4 and the I.H. for $|x|$ times.
2. By part 1, with nx^k_M as the f.
3. If we have $M[x] = F(x)\ [k|x|^c]$ for some F and M, then, by part 2, F can be computed in $\mathcal{PAL}_c$. By this same argument, the left inclusion of theorem 13.2 follows by next lemma.

Lemma 18 For all $e \in \mathcal{PAL}_0$ and λ there is $e_\lambda \in \mathcal{PAL}_\lambda$ such that

$$e_\lambda[x] = e \uparrow_{\geq} H_\lambda(n)[x] \qquad (n := |x| \geq 2).$$

Construction. 1. Given e, we claim that there is a quadratic time TM M_1 such that

$$\begin{array}{lrcl} a & M_1[y;(x,\underline{l},\omega^*)] & = & (x,(e\ \mathrm{ite}^{l+n})^*) \\ b & M_1[y;(x,\underline{l},\mu+\omega^*)] & = & ((x,\underline{2(n+l)},\mu^*),(nx_{M_y}\ \mathrm{ite}^{2(n+l)}\ \mu)^*) \\ c & M_1[y;(x,\underline{l},\lambda^*)] & = & ((x,\underline{2(n+l)},\lambda[2(n+l)]^*),(nx_{M_y}\ \mathrm{ite}^{2(n+l)}\ \lambda[2(n+l)])^*). \end{array}$$

Indeed, in the worst case (line c), M_1 has to copy its input by infixing meanwhile the output of FS and NX; since the calls to these TM's are not nested, its time complexity is quadratic.
2. By Note 16.1 there is a quadratic time TM $M!$ such that $M![z] = M_1[M!^*, z]$.
3. Let $(c_0+n)^2$ bound above the time complexity of $M!$, and define (for $c := c_0 + |\lambda^*|$)

$$e^*_\lambda := ((Q,(\underline{c},\lambda^*),COR),(nx\ \mathrm{ite}^c\ \lambda)^*,SB) \qquad (nx := nx_{M!}).$$

Proof. Observe first that we have $x\ x\ e^*_\lambda \Rrightarrow$

$$\begin{array}{llll} \mathrm{A} \Rrightarrow & x & ((\mathrm{cons^R}_{(\underline{c},\lambda^*)})^*,(nx\ \mathrm{ite}^c\ \lambda)^*) & \text{prod. 2, definition of } e_\lambda \\ \mathrm{B} \Rrightarrow & (x,\underline{c},\lambda^*) & (nx\ \mathrm{ite}^c\ \lambda)^* & \text{prod. 1.} \end{array}$$

The result follows since we claim that for all $l \geq c$ we have

$$x \quad (x,\underline{l},\lambda^*) \quad (nx\ \mathrm{ite}^l\ \lambda)^* \Rrightarrow x \uparrow_{\geq} H_\lambda(n+l) \quad \mathrm{NIL}. \tag{1}$$

Induction on λ, after observing that runtime for $M![(x,\underline{l},\lambda^*)]$ is $(n+l+c)^2 < n^l$.
Basis. $\lambda = \omega$. We have $\quad x \quad (x,\underline{l},\omega^*) \quad (nx\ \mathrm{ite}^l\ \omega)^* \Rrightarrow$

$$\begin{array}{llll} \mathrm{C} \Rrightarrow & M![(x,\underline{l},\omega^*)] & \omega^* & \text{prod. 5, def. of } nx \text{ and } c\text{, Lemma 17.2} \\ \mathrm{D} \Rrightarrow & (x,(e\ \mathrm{ite}^{n+l})^*) & \omega^* & \text{line } a \\ \mathrm{E} \Rrightarrow & x & (e\ \mathrm{ite}^{n+l})^* & \text{prod. 6, since } o[(e\ \mathrm{ite}^{n+l})^*] < \omega. \end{array}$$

The claim (1) follows by Lemma 17, since we obviously have $n^{n+l} > H_\omega(n) = 2n$.
Step. Define $k := 2(n+l)$. Case 1. $\lambda = \mu+\omega$. We have $\quad x \quad (x,\underline{l},\mu+\omega^*) \quad (nx\ \mathrm{ite}^l\ \mu+\omega)^* \Rrightarrow$

$$\begin{array}{llll} \mathrm{C}_1 \Rrightarrow & M![(x,\underline{l},\mu+\omega^*)] & \mu+\omega^* & \text{like line C} \\ \mathrm{D}_1 \Rrightarrow & (x,\underline{k},\mu^*) & (nx\ \mathrm{ite}^k\ \mu)^* & \text{line } b\text{, prod. 6, since } \mu < \lambda. \end{array}$$

The claim (1) follows by Lemma 17 and I.H., since it is known that $H_{\mu+\omega}(n+l) = H_\mu(k)$.
Case 2. Not $\lambda =_{nf} \mu+\omega$. We have $\quad x \quad (x,\underline{l},\lambda^*) \quad (nx\ \mathrm{ite}^l\ \lambda)^* \Rrightarrow$

$$\begin{array}{llll} \mathrm{C}_2 \Rrightarrow & M![(x,\underline{l},\lambda^*)] & \lambda^* & \text{like line C} \\ \mathrm{D}_2 \Rrightarrow & (x,\underline{k},\lambda[k]^*) & (nx\ \mathrm{ite}^k\ \lambda[k])^* & \text{like D}_1\text{, after replacing ``\textit{line b}'' with ``\textit{line c}''.} \end{array}$$

The claim (1) follows by Lemma 17 and I.H., since $H_\lambda(m) < H_{\lambda[2m]}(m)$.

5 Simulation by TM's

A k-tapes TM IN ($k > 3$), can be defined, which plays the role of *interpreter* for $\mathcal{PAL}_\alpha$. While x, y, z are stored in its tapes X, Y, Z, IN simulates the productions carried out by R_3 when the same values x, y, z are stored in registers I, S, P. We write IN : $x\ y\ w \models u\ z \quad [\tau; \sigma]$ (τ or σ sometimes omitted) to mean that IN replaces y and w by u and z in time τ and in space σ.

Lemma 19 1. For all $g \in \mathcal{PAL}_0$ and z we have

$$\text{IN} : x\ y\ (g^*, z) \models g[y]\ z \quad [K|g||y|; |g|] \qquad \text{for a constant } K.$$

2. Assume IN : $x\ y\ (g^*, z) \models g[y]\ z \quad [K|g||y|\tau(n); |g|\sigma(n)] \qquad (n := |x|)$.
We have IN: $x\ y\ (f^*, z) \models f[y]\ z \quad [Km|y|\tau(n)n; |g|\sigma(n)n] \qquad (f = g\,\text{ite};\ m := |f|)$.

3. Hence for all $f \in \mathcal{TP}$ we have IN : $x\ y\ (f^*, z) \models f[y]\ z \quad [K|y|n^m; n^m]$.

Proof. 1. Induction on the definition of g, after noting that parsing y is a linear-time task.

2. IN needs time $\leq Kn|g|$ (for prod. 4) $+ Kn|g||y|\tau(n)$ (for n simulations of g)
$\leq Kn|g|(|y|\tau(n) + 1) \leq Knm|y|\tau(n)$ (since $|g| < m$).

Similarly for the part about space.
3. Assume $f = e\,\text{ite}^c$ ($e \in \mathcal{PAL}_0$). By parts 1 and 2 we have $|f[x]| \leq |e|n^c \leq n^{|e|+c}$.

Next note prepares the evaluation of the complexity at the transfinite of IN.

Note 20 1. By induction on λ and cases with respect to $\lambda =_{nf} \beta + \omega^\gamma$ one proves that

$$\alpha < \lambda \text{ and } |\alpha| < n^m \text{ implies } \alpha < \lambda[n^m].$$

2. From part 1, Lemma 19.3, and clause "$g \in \mathcal{TP}$" of Def. 7 we obtain for all $f \in \mathcal{PAL}_\lambda$

$$f[x] = h[x] \text{ for some } h \in \mathcal{PAL}_{\lambda[n^m]} \text{ such that } |h| \leq n^m \quad (n := |x|,\ m := |f|).$$

Lemma 21 $\mathcal{PAL}_\alpha \subseteq \mathcal{E}_\alpha$ for all $\alpha \geq 3$.

Proof. We show by induction on α that for all $x, y, z, f \in \mathcal{PAL}_\alpha$ ($\omega \leq \alpha$) we have

$$\text{IN} : x\ y\ (f^*, z) \models f[y] \quad [K|y|E_\alpha(n_2(m^2)); E_\alpha(n_2(m^2))] \qquad (n := |x|,\ m := |f|), \quad (1)$$

where $n_0(l) := l$; $n_{c+1}(l) := n^{n_c(l)}$. (The result follows, for $y = x$ and z absent, since $\mathcal{E}_\alpha$ ($3 \leq \alpha$) is honest with respect to time, and since $KnE_\alpha(n_2(m^2)) \leq E_\alpha^{K+m+4}(n) \in \mathcal{E}_\alpha$.)

Basis. $\alpha = \omega$. The form of f is $g\,\omega$ with $g \in \mathcal{TP}$. We have IN : $x\ y\ (f^*, z)\ \models$

$$\begin{array}{lll} \models\ y\ (g^*, \omega^*, z) & [Km; 0] & \text{prod. 5} \\ \models\ g[y]\ (\omega^*, z) & [K|y|n_2(m); n_2(m)] & \text{Lemma 19.3 and Note 20.2} \end{array}$$

Step. We may assume that α is the limit $\lambda =_{nf} \beta + \omega^\gamma$ (since else the result follows by I.H. and Lemma 19.2). Hence the form of f is $g\,\lambda$ with $g \in \mathcal{TP}$. Case 1. $\gamma > 1$. By prod. 5 and Lemma 19.3 we have

$$\text{IN} : x\ y\ (f^*, z)\ \models\ g[y]\ (\lambda^*, z) \quad [(K+1)|y|n^m; n^m]. \tag{2}$$

By Note 20.2, we have $g[y] = (u, w)$ with w in the form $(h\,\lambda[n^m])^*$. Hence we obtain

$$\text{IN} : x\ y\ (f^*, z)\ \models\ u\ ((h\,\lambda[n^m])^*, z) \quad [(K+1)|y|n^m; n^m] \qquad \text{(prod. 6 and 5)}. \tag{3}$$

By the I.H. (with $h\,\lambda[n^m]$ as f and with $\lambda[n^m]$ as α), since $n_2((n^m)^2) = n_3(2m)$, we obtain

$$\text{IN} : x\ u\ ((h\,\lambda[n^m])^*, z)\ \models\ (h\,\lambda[n^m])[u]\ z \quad [K|u|E_{\lambda[n^m]}(n_3(2m)); E_{\lambda[n^m]}(n_3(2m))]. \tag{4}$$

By summing-up the amounts of time (2)-(4) we obtain (since $|u| \leq n^m$)

$$(K+1)|y|n^m + (K+1)|y|n^m + Kn^m E_{\lambda[n^m]}(n_3(2m)) \leq KE_{\lambda[n^m]}(n_3(2m+1)). \tag{5}$$

From (4) we obtain (1) since we have (recall that we have $E_3(l) > l^l$ and $\lambda \geq \omega^2$)

$$E_{\lambda[n^m]}(n_3(2m+1)) \leq E_{\lambda[n^m]}(E_3(n_2(2m+1))) \leq E_{\lambda[n^m+1]}(n_2(2m+1)) \leq E_{\lambda[n_2(m^2)]}(n_2(m^2)).$$

Case 2. $\alpha = \mu + \omega$. We obtain (1) by arguments like under Case 1, after observing that $E_{\lambda[n^m]}(n_3(2m+1)) \leq E_{\mu+n^m}(E_3(n_2(2m+1))) \leq E_{\mu+n^m+1}(n_2(2m+1)) \leq E_{\lambda[n_2(m^2)]}(n_2(m^2))$.

References

[1] S. Caporaso, E. Covino, G. Pani, *A predicative approach to the classification problem.* J. Funct. Programming 11(2001)95-116.

[2] D. Leivant, *Intrinsic Theories and Computational Complexity*, in D. Leivant (ed.), Logic and Computational Complexity, LNCS 960(1995), 177-194, Springer.

[3] D. Leivant, *Intrinsic reasoning about functional programs I: fist-order theories.* To appear in ??? (papers dedicated to A. S. Troelstra on his 60th birthday).

[4] A. Weiermann, *What makes a (pointwise) subrecursive function slow growing?* In S. B. Cooper and J.K.Truss (Eds) Sets and Proofs. London math. soc. lecture notes 258(1999), Cambridge U. Press.

The strength of non-size increasing computation

Martin Hofmann[*]

Abstract

We study the expressive power non-size increasing recursive definitions over lists. This notion of computation is such that the size of all intermediate results will automatically be bounded by the size of the input so that the interpretation in a finite model is sound with respect to the standard semantics. Many well-known algorithms with this property such as the usual sorting algorithms are definable in the system in the natural way. The main result is that a characteristic function is definable if and only if it is computable in time $O(2^{p(n)})$ for some polynomial p.

The method used to establish the lower bound on the expressive power also shows that the complexity becomes polynomial time if we allow primitive recursion only. This settles an open question posed in [1, 6].

The key tool for establishing upper bounds on the complexity of derivable functions is an interpretation in a finite relational model whose correctness with respect to the standard interpretation is shown using a semantic technique.

1 Introduction

Consider the following recursive definition of a function on lists:

$$\begin{aligned}&\mathtt{twice}(\mathsf{nil}) = \mathsf{nil}\\ &\mathtt{twice}(\mathsf{cons}(x,l)) = \mathsf{cons}(\mathsf{tt},\mathsf{cons}(\mathsf{tt},\mathtt{twice}(l)))\end{aligned}\tag{1}$$

Here nil denotes the empty list, $\mathsf{cons}(x,l)$ denotes the list with first element x and remaining elements l. tt,ff are the members of a type T of truth values. We have that $\mathtt{twice}(l)$ is a list of length $2\cdot|l|$ where $|l|$ is the length of l. Now consider

$$\begin{aligned}&\mathtt{exp}(\mathsf{nil}) = \mathsf{cons}(\mathsf{tt},\mathsf{nil})\\ &\mathtt{exp}(\mathsf{cons}(x,l)) = \mathtt{twice}(\mathtt{exp}(l))\end{aligned}\tag{2}$$

We have $|\mathtt{exp}(l)| = 2^{|l|}$ and further iteration leads to elementary growth rates.

This shows that innocuous looking recursive definitions can lead to enormous growth. In order to prevent this from happening it has been suggested in [2, 9] to rule out definitions like (2) above, where a recursively defined function, here `twice`, is applied to the result of a recursive call. Indeed, it has been shown that such discipline restricts the definable functions to the polynomial-time computable ones and moreover every polynomial-time computable *function* admits a definition in this style.

[*]Fachbereich Mathematik, TU Darmstadt,Schlossgartenstr. 7, 64289 Darmstadt, Germany, mhofmann@mathematik.tu-darmstadt.de

Many naturally occurring *algorithms*, however, do not fit this scheme. Consider, for instance, the definition of insertion sort:

$$\begin{aligned}
&\texttt{insert}(x,\mathsf{nil}) = \mathsf{cons}(x,\mathsf{nil})\\
&\texttt{insert}(x,\mathsf{cons}(y,l)) = \mathsf{if}\ x \leq y\ \mathsf{then}\ \mathsf{cons}(x,\mathsf{cons}(y,l))\ \mathsf{else}\ \mathsf{cons}(y,\texttt{insert}(x,l))\\
&\texttt{sort}(\mathsf{nil}) = \mathsf{nil}\\
&\texttt{sort}(\mathsf{cons}(x,l)) = \texttt{insert}(x,\texttt{sort}(l))
\end{aligned} \tag{3}$$

Here just as in (2) above we apply a recursively defined function (insert) to the result of a recursive call (sort), yet no exponential growth arises.

It has been argued in [3] and [6] that the culprit is definition (1) because it defines a function that increases the size of its argument and that non size-increasing functions can be arbitrarily iterated without leading to exponential growth.

In [3] a number of partly semantic criteria were offered which allow one to recognise when a function definition is non size-increasing. In [6] we have given syntactic criteria based on linearity (bound variables are used at most once) and a so-called resource type $\diamond$ which counts constructor symbols such as "cons" on the left hand side of an equation.

This means that cons becomes a ternary function taking one argument of type $\diamond$, one argument of some type A (the head) and a third argument of type $\mathsf{L}(A)$, the tail. There being no closed terms of type $\diamond$ the only way to apply cons is within a recursive definition; for instance, we can write

$$\begin{aligned}
&\mathsf{append}(\mathsf{nil},l_2) = l_2\\
&\mathsf{append}(\mathsf{cons}(d,a,l_1),l_2) = \mathsf{cons}(d,a,\mathsf{append}(l_1,l_2)
\end{aligned} \tag{4}$$

Alternatively, we may write

$$\mathsf{append}(l_1,l_2) = \mathsf{match}\ l\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}l_2 \mid \mathsf{cons}(d,a,l_1'){\Rightarrow}\mathsf{cons}(d,\mathsf{append}(l_1,l_2) \tag{5}$$

We notice that the following attempted definition of twice is illegal as it violates linearity (the bound variable d is used twice):

$$\begin{aligned}
&\texttt{twice}(\mathsf{nil}) = \mathsf{nil}\\
&\texttt{twice}(\mathsf{cons}(d,x,l)) = \mathsf{cons}(d,\mathtt{tt},\mathsf{cons}(d,\mathtt{tt},\texttt{twice}(l)))
\end{aligned} \tag{6}$$

The definition of insert, on the other hand, is in harmony with linearity provided that insert gets an extra argument of type $\diamond$ and, moreover, we assume that the inequality test returns its arguments for subsequent use.

The main result of [6] and [1] was that all functions thus definable by *structural recursion* are polynomial-time computable even when higher-order functions are allowed. In [7] it has been shown that general-recursive first-order definitions admit a translation into a fragment of the programming language C without dynamic memory allocation ("malloc") which on the one hand allows one to automatically construct imperative implementations of algorithms on lists which do not require extra space or garbage collection. More precisely, this translation maps the resource type $\diamond$ to the C-type `void *` of pointers. The cons function is translated into the C-function which extends a list by a given value using a provided piece of memory. It is proved that the pointers arising as denotation of terms of type $\diamond$ always point to free memory space which can thus be safely overwritten.

This translation also demonstrates that all definable functions are computable on a Turing machine with linearly bounded work tape and an unbounded stack (to accommodate general recursion) which by a result of Cook[1] [4] equals the complexity class $DTIME(2^{O(n)})$. It was also shown in [7] that any such function admits a representation.

In the presence of higher-order functions the translation into C breaks down as C does not have higher-order functions. Of course, higher-order functions can be simulated as closures, but this then requires arbitrary amounts of space as closures can grow proportionally to the runtime. In a system based on structural recursion such as [6] this is not a problem as the runtime is polynomially bounded there. The hitherto open question of complexity of general recursion with higher-order functions is settled in this paper and shown to require a polynomial amount of space only in spite of the unbounded runtime.

We thus demonstrate that a function is representable with general recursion and higher-order functions iff it is computable in polynomial space and an unbounded stack or equivalently (by Cook's result) in time $O(2^{p(n)})$ for some polynomial p. The lower bound of this result also demonstrates that indeed all characteristic functions of problems in P are definable in the structural recursive system. This settles a question left open in [1, 6].

In view of the results presented in this paper, these systems of non size-increasing computation thus provide a very natural connection between complexity theory and functional programming. There is also a connection to finite model theory in that—as will be shown below—programs admit a sound interpretation in a finite model. This improves upon earlier combinations of finite model theory with functional programming [5] where interpretation in a finite model was achieved in a brute-force way by changing the meaning of constructor symbols, e.g. successor of the largest number N was defined to be N itself. In those systems it is the responsibility of the programmer to account for the possibility of cut-off when reasoning about the correctness of programs. In the systems studied here linearity and the presence of the resource types automatically ensure that cutoff never takes place. Formally, it is shown that the standard semantics in an infinite model agrees with the interpretation in a certain finite model for all well-formed programs.

Another piece of related work is Jones' [8] where the expressive power of cons-free higher-order programs is studied. It is shown there that first-order cons-free programs define polynomial time , whereas second-order programs define EXPTIME. This shows that the presence of "cons", tamed by linearity and the resource type changes the complexity-theoretic strength. While loc. cit. also involves Cook's abovementioned result (indeed, this result was brought to the author's attention by Neil Jones) the other parts of the proof are quite different.

[1]This result asserts that if $L(n) > \log(n)$ then $DTIME(2^{O(L(n))})$ equals the class of functions computable by a Turing machine with an $L(n)$-bounded R/W-tape and an unbounded stack.

2 Syntax and typing rules

The terms of the languag are given by the following grammar:

$e ::=$	x	variable
$\mid$	$f(e_1,\dots,e_n)$	function application
$\mid$	tt,ff	boolean constant
$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e''$	conditional
$\mid$	$e_1 \otimes e_2$	pairing
$\mid$	nil	empty list
$\mid$	$\mathsf{cons}(e_1,e_2,e_3)$	cons with res. arg.
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}e_2 \mid \mathsf{cons}(d,h,t){\Rightarrow}e_3$	list elimination
$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ x\otimes y{\Rightarrow}e_2$	pair elim.
$\mid$	$\lambda x.e$	linear lambda abstraction
$\mid$	$e_1 e_2$	linear function application

The match constructs as well as λ bind variables.

The *types* are given by the following grammar.

$$A ::= \mathsf{T} \mid \diamond \mid \mathsf{L}(A) \mid A_1 \otimes A_2 \mid A_1 \multimap A_2$$

Here T is the type of truth values, $\mathsf{L}(A)$ stands for lists with entries of type A, $A_1 \otimes A_2$ is the type of pairs with first component of type A_1 and second component of type A_2. The type $A_1 \multimap A_2$ is the type of functions from A_1 to A_2, and finally $\diamond$ is the resource type. The *heap-free* types contain T and are closed under $\otimes$. Variables of heap-free type may be used more than once as described by rule CONTR below.

In [7] also tree types and disjoint union types were considered. We refrain from doing so here for the sake of simplicity. However, it has been checked that all the constructions presented here carry over to this richer setting.

A *signature* Σ maps a finite set of function symbols to expressions of the form $(A_1,\dots,A_n)\to B$ where $A_1 \dots A_n$ and B are types.

A *typing context* Γ is a finite function from variables to types; if $x \notin \mathrm{dom}(\Gamma)$ then we write $\Gamma, x{:}A$ for the extension of Γ with $x \mapsto A$. More generally, if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ then we write Γ, Δ for the disjoint union of Γ and Δ. If such notation appears in the premise or conclusion of a rule below it is implicitly understood that these disjointness conditions are met. We write $e[x/y]$ for the term obtained from e by replacing all occurrences of the free variable y in e by x after suitable renaming of bound variables so as to prevent capture. We consider terms modulo renaming of bound variables.

Let Σ be a signature. The *typing judgment* $\Gamma \vdash_\Sigma e : A$ read "expression e has type A in typing context Γ and signature Σ" is defined by the following rules.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_\Sigma x : \Gamma(x)} \qquad (\textsc{Var})$$

$$\frac{\Sigma(f) = (A_1,\dots,A_n)\to B \qquad \Gamma_i \vdash_\Sigma e_i : A_i \text{ for } i = 1\dots n}{\Gamma_1,\dots,\Gamma_n \vdash_\Sigma f(e_1,\dots,e_n) : B} \qquad (\textsc{Sig})$$

$$\frac{\Gamma, x{:}A, y{:}A \vdash_\Sigma e : B \qquad A \text{ heap-free}}{\Gamma, x{:}A \vdash_\Sigma e[x/y] : B} \qquad (\textsc{Contr})$$

$$\frac{c \in \{\mathsf{tt}, \mathsf{ff}\}}{\Gamma \vdash_\Sigma c : \mathsf{T}} \qquad (\textsc{Const})$$

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{T} \qquad \Delta \vdash_\Sigma e' : A \qquad \Delta \vdash_\Sigma e'' : A}{\Gamma, \Delta \vdash_\Sigma \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A} \qquad (\textsc{If})$$

$$\frac{\Gamma \vdash_\Sigma e : A \qquad \Delta \vdash_\Sigma e' : B}{\Gamma, \Delta \vdash_\Sigma e \otimes e' : A \otimes B} \qquad (\textsc{Pair})$$

$$\frac{\Gamma \vdash_\Sigma e : A \otimes B \qquad \Delta, x{:}A, y{:}B \vdash_\Sigma e' : C}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ x \otimes y \Rightarrow e' : C} \qquad (\textsc{Split})$$

$$\Gamma \vdash_\Sigma \mathsf{nil} : \mathsf{L}(A) \qquad (\textsc{Nil})$$

$$\frac{\Gamma_d \vdash_\Sigma e_d : \Diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)} \qquad (\textsc{Cons})$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma e : \mathsf{L}(A) \\ \Delta \vdash_\Sigma e_{\mathsf{nil}} : B \\ \Delta, d{:}\Diamond, h{:}A, t{:}\mathsf{L}(A) \vdash_\Sigma e_{\mathsf{cons}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}} : B} \qquad (\textsc{List-Elim})$$

$$\frac{\Gamma, x{:}A \vdash_\Sigma e : B}{\Gamma \vdash \lambda x.e : A \multimap B} \qquad (\textsc{Lam})$$

$$\frac{\Gamma \vdash_\Sigma e_1 : A \multimap B \qquad \Delta \vdash e_2 : A}{\Gamma, \Delta \vdash_\Sigma e_1 e_2 : B} \qquad (\textsc{App})$$

Application of function symbols is linear in the sense that several operands must in generalnot share common free variables. This is because of the implicit side condition on juxtaposition of contexts mentioned above. In view of rule Contr, however, variables of a heap-free type may be shared and moreover thesame free variable may appear in different branches of a case distinction as follows e.g. from the form of rule If. It follows by standard type-theoretic techniques that type checking for this system is decidable in linear time. More precisely, we have a linear time computable function which given a context Γ, a term e in normal form[2], and a type A either returns a minimal subcontext Δ of Γ such that $\Delta \vdash e : A$ or returns "failure" in the case where $\Gamma \vdash e : A$ does not hold. This function can be defined by primitive recursion over e.

A *program* consists of a signature Σ and for each symbol $f : (A_1, \dots, A_n) \to B$ contained in Σ a term e_f such that $x_1{:}A_1, \dots, x_n{:}A_n \vdash_\Sigma e_f : B$.

[2]i.e. one that does not contain instance of match applied to constructors ($\mathsf{nil}, \mathsf{cons}, \otimes$) or λ-abstractions in applied position

3 Denotational semantics

In order to specify the purely functional meaning of programs we introduce a denotational semantics following [10].

A partially ordered set $D = (D, \leq)$ is a *complete partial order*, cpo for short, if each increasing chain $x_0 \leq x_1 \leq \dots$ has a least upper bound $\bigvee_i x_i$ in D. A function from cpo D to cpo E is continuous if it is monotone and preserves these least upper bounds. Any set forms a (discrete) cpo. If D is a cpo its lifting $D_\perp$ is formed by freely adjoining a least element $\perp$. For cpos D and E we have their cartesian product $D \times E$ with the component-wise ordering. We write (x, y) for the pair with components x and y and if $p = (x, y)$ we write $p.1 = x$ and $p.2 = y$ for the first and second projections. We assume that $\times$ associates to the right so that e.g. the second component of $p \in D \times E \times F$ is obtained as $p.2.1$. We have the continuous function space $D \to E$ consisting of continuous functions from D to E with the point-wise ordering. Elements of $D \to E$ may be defined using λ-notation if continuity is ensured. For instance, if $e \in E$ the expression $\lambda x.e$ denotes the constant function in $D \to E$.

The cpo $\mathsf{L}(D)$ consists of finite lists of elements of D with lists of equal length ordered component-wise and lists of different length being incomparable. We use the notation [] for the empty list, $a :: l$ for the list with first element a and remaining elements l, we write $[a_1, \dots, a_n]$ for the list with members $a_1, \dots, a_n$ and $l_1 @ l_2$ for the concatenation of lists l_1 and l_2. We write $|l|$ for the length of a list l.

We assign a cpo to each type by

$$\begin{array}{lll} [\![\mathsf{T}]\!] = \{\mathsf{tt}, \mathsf{ff}\} & [\![\diamond]\!] = \{0\} & [\![\mathsf{L}(A)]\!] = \mathsf{L}([\![A]\!]) \\ [\![A \otimes B]\!] = [\![A]\!] \times [\![B]\!] & [\![A \multimap B]\!] = [\![A]\!] \to [\![B]\!]_\perp & \end{array}$$

To each program $P = (\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$ we can now associate a mapping $[\![P]\!]$ such that $[\![P]\!](f)$ is a continuous map from $[\![A_1]\!] \times \dots \times [\![A_n]\!]$ to $[\![B]\!]_\perp$ for each $f : (A_1, \dots, A_n) \to B$.

This meaning is given in the standard fashion as the least fixpoint of an appropriate compositionally defined operator, as follows.

A *valuation* of a context Γ is a function η such that $\eta(x) \in [\![\Gamma(x)]\!]$ for each $x \in \mathrm{dom}(\Gamma)$; a valuation of a signature Σ is a function ρ such that $\rho(f) \in [\![A_1]\!] \times \dots \times [\![A_n]\!] \to [\![B]\!]_\perp$ whenever $f \in \mathrm{dom}(\Sigma)$.

To each expression e such that $\Gamma \vdash_\Sigma e : A$ we assign a function mapping a valuation η of Γ and a valuation ρ of Σ to an element $[\![e]\!]_{\eta,\rho} \in [\![A]\!]$ in the obvious way, i.e. function symbols and variables are interpreted according to the valuations; basic functions and expression formers are interpreted by the eponymous set-theoretic operations, ignoring the arguments of type $\diamond$ in the case of constructor functions. The formal definition of $[\![-]\!]_{\eta,\rho}$ is by induction on terms.

Here are a few representative clauses.

$$\begin{aligned}
&[\![x]\!]_{\eta,\rho} = \eta(x)\\
&[\![f(e_1,\dots,e_n)]\!]_{\eta,\rho} = \rho(f)([\![e_1]\!]_{\eta,\rho},\dots,[\![e_n]\!]_{\eta,\rho})\\
&[\![\mathsf{cons}(e_1,e_2,e_3)]\!]_{\eta,\rho} = [\![e_2]\!]_{\eta,\rho} :: [\![e_3]\!]_{\eta,\rho}\\
&[\![\mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}e_1 \mid \mathsf{cons}(d,h,t){\Rightarrow}e_2]\!]_{\eta,\rho}\\
&\quad = [\![e_1]\!]_{\eta,\rho}\\
&\quad \text{when } [\![e]\!]_{\eta,\rho} = [] \text{ and}\\
&\quad = [\![e_2]\!]_{\eta[d\mapsto 0,h\mapsto v_h,t\mapsto v_t],\rho}\\
&\quad \text{when } [\![e]\!]_{\eta,\rho} = v_h :: v_t\\
&[\![\lambda x.e]\!]_{\eta,\rho}(v) = [\![e]\!]_{\eta[x\mapsto v],\rho}\\
&[\![e_1 e_2]\!]_{\eta,\rho} = [\![e_1]\!]_{\eta,\rho}([\![e_2]\!]_{\eta,\rho})
\end{aligned}$$

A *program* $(\Sigma,(e_f)_{f\in\mathrm{dom}(\Sigma)})$ is interpreted as the least upper bound of the following (pointwise) increasing sequence of valuations: $\rho_0(f)(\vec{v}) = \bot$ and

$$\rho_{i+1}(f)(v_1,\dots,v_n) = [\![e_f]\!]_{\rho_i,\eta} \tag{7}$$

where $\eta(x_i) = v_i$, for any $f \in \mathrm{dom}(\Sigma)$. Notice that $\rho = \bigvee_i \rho_i$ satisfies

$$\rho(f)(v_1,\dots,v_n) = [\![e_f]\!]_{\rho,\eta} \tag{8}$$

and is minimal with this property.

We stress that this order-theoretic semantics does not say anything about computational complexity. Its *only* purpose is to pin down the functional denotations of programs so that we can formally state what it means to implement a function. Accordingly, the resource type is interpreted as a singleton set, $\otimes$ and $\multimap$ are interpreted as ordinary product and function space disregarding linearity.

If f is a function symbol in defined in a program P that is clear from the surrounding context then we may abbreviate $[\![P]\!](f)$ to $[\![f]\!]$.

3.1 Examples

Reverse:

```
$\texttt{rev\_aux} : (\mathsf{L}(\mathsf{N}),\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$
$\texttt{reverse} : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$
$e_{\texttt{rev\_aux}}(l, acc) = \mathsf{match}\ l\ \mathsf{with}\ \mathsf{nil}{\Rightarrow}acc \mid \mathsf{cons}(d,h,t){\Rightarrow}\texttt{rev\_aux}(t,\mathsf{cons}(d,h,acc))$
$e_{\texttt{reverse}}(l) = \texttt{rev\_aux}(l,\mathsf{nil})$
```

Insertion sort

```
$\texttt{insert} : (\diamond,\mathsf{N},\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$
$\texttt{sort} : (\mathsf{L}(\mathsf{N}))\to\mathsf{L}(\mathsf{N})$
$e_{\texttt{insert}}(d,a,l) = \mathsf{match}\ l\ \mathsf{with}$
    $\mathsf{nil}{\Rightarrow}\mathsf{nil}$
    $\mid \mathsf{cons}(d',b,t){\Rightarrow}\mathsf{if}\ a \le b$
        $\mathsf{then}\ \mathsf{cons}(d,a,\mathsf{cons}(d',b,t))$
        $\mathsf{else}\ \mathsf{cons}(d,b,\texttt{insert}(d',a,t))$
$e_{\texttt{sort}}(l) = \mathsf{match}\ l\ \mathsf{with}$
    $\mathsf{nil}{\Rightarrow}\mathsf{nil}$
    $\mid \mathsf{cons}(d,a,t){\Rightarrow}\texttt{insert}(d,a,\texttt{sort}(t))$
```

Apply a function to the tail of a list

```
AppTail : $(A \multimap A, \mathsf{L}(A)) \to \mathsf{L}(A)$
$e_{\texttt{AppTail}}(f, a, l) = \mathsf{match}\ l\ \mathsf{with}$
      $\mathsf{nil} \Rightarrow \mathsf{nil}$
      $\mid \mathsf{cons}(d, b, t) \Rightarrow \mathsf{match}\ t\ \mathsf{with}\ \mathsf{nil} \Rightarrow \mathsf{cons}(d, f(b), \mathsf{nil})$
        $\mid \mathsf{cons}(d', b', t') \Rightarrow \texttt{AppTail}(\mathsf{cons}(d', b', t'))$
```

Composing all functions in a list

```
ComposeList : $(\mathsf{L}((\Diamond \otimes A) \multimap A)) \to A \multimap A$
$e_{\texttt{ComposeList}}(l, a) = \mathsf{match}\ l\ \mathsf{with}$
      $\mathsf{nil} \Rightarrow \lambda a.a$
      $\mid \mathsf{cons}(d, f, t) \Rightarrow \lambda a.f(d \otimes \texttt{ComposeList}(t)(a))$
```

Higher-order tail recursion

```
Contrived : $(A, A \multimap A) \to A$
$e_{\texttt{Contrived}}(x, f) = \mathsf{if}\ \mathsf{p}(x)\ \mathsf{then}\ f(x)$
      $\mathsf{else\ if}\ \mathsf{q}(x)\ \mathsf{then}\ \texttt{Contrived}(\mathsf{a}(x), \lambda y.\mathsf{g}(f(\mathsf{g}(x))))$
          $\mathsf{else}\ \texttt{Contrived}(\mathsf{b}(x), \lambda y.\mathtt{h}(f(\mathtt{h}(x))))$
```

In the last example, $\mathsf{p}, \mathsf{q} : (A) \to \mathsf{T}$ and $\mathsf{a}, \mathsf{b}, \mathsf{g}, \mathtt{h} : (A) \to A$ are arbitrary function symbols defined independently or indeed simultaneously with `Contrived`. The point of the example is that under a functional evaluation strategy the intermediate term denoting the currently accumulated function grows arbitrarily. Many more examples are given in [6, 7].

4 Expressivity

In this section we characterise the functions of type $(\mathsf{L}(\mathsf{T})) \to \mathsf{L}(\mathsf{T})$ definable in the system. We will say nothing about higher-order functionals definable in the system, notice, however, that a first-order function may involve a higher-order functional as part of its definition. This situation is encompassed by our characterisation.

Let us write W for the type $\mathsf{L}(\mathsf{T})$ and T for the set $\{\mathtt{tt}, \mathtt{ff}\}$ and W for the set $T^* = [\![\mathsf{L}(\mathsf{T})]\!] = [\![\mathsf{W}]\!]$. For a set A we define $\mathsf{L}_n(A) = \{w \in A^* \mid |w| = n\}$ as the set of lists of length n over A. We write $W_n = \mathsf{L}_n(T)$ so that $W_n \subseteq W$. Elements of W_n will be identified with the set $\{0, \ldots, 2^n - 1\}$ using the binary encoding. E.g. $W_5 \ni [\mathtt{ff}, \mathtt{tt}, \mathtt{tt}, \mathtt{ff}, \mathtt{ff}] = 12$.

If $A_1, \ldots, A_n, B$ are types and $f : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to [\![B]\!]_\perp$ is a function then we say that f is *representable* if there exists a program containing a function symbol $\mathtt{f} : (A_1, \ldots, A_n) \to B$ such that $[\![\mathtt{f}]\!] = f$. Our aim in this section is to prove the following result.

Theorem 4.1 *Let $f : W \to W$ be a function such that $|f(w)| \leq |w|$ and such that $f(x)$ is computable in time $O(2^{p(|x|)})$ for some polynomial p. Then f is representable.*

Definition 4.2 Let $s : \mathbb{N} \to \mathbb{N}$ be a function with $s(n) < 2^n$ and $k \in \mathbb{N}$ be a number. A (k, s)-*storage device* is given by the following data:

- a set $S = [\![\mathsf{S}]\!]$ for some type S

- a family of subsets $S_n \subseteq S$ for $n \in \mathbb{N}$.
- a representable function (a constant) $\mathit{init} :\to S$, i.e. there is $\texttt{init} : ()\to\texttt{S}$ with $[\![\texttt{init}]\!] = \mathit{init}$,
- a representable function $\mathit{read} : W \times W \times S \to W \times W \times T \times S$, i.e., there is $\texttt{read} : (\mathsf{L}(\mathsf{T}), \mathsf{L}(\mathsf{T}), \mathsf{S})\to\mathsf{L}(\mathsf{T}) \otimes \mathsf{L}(\mathsf{T}) \otimes \mathsf{T} \otimes \mathsf{S}$ with $[\![\texttt{read}]\!] = \mathit{read}$,
- a representable function $\mathit{write} : W \times W \times T \times S \to W \times W \times S$, i.e., there is ...

such that for all $n \in \mathbb{N}$ and $w, w_1, w_2, w_3 \in W_{kn}$, $a, a' \in W_n$ and $s \in S_n$ the following are satisfied:

- $\mathit{init}() \in S_n$
- $\mathit{read}(w, a, s) = (w', a', b, s')$ implies $w' \in W_{kn}, a' \in W_n, s' = s$
- $\mathit{write}(w, a, b, s) = (w', a', s')$ implies $w' \in W_{kn}, a' \in W_n, s' \in S_n$
- $\mathit{read}(w_1, a, \mathit{write}(w_2, a, b, s).2.2) = b$ provided that $a < s(n)$
- $\mathit{read}(w_1, a, \mathit{write}(w_2, a', b, s).2.2) = \mathit{read}(w_3, a, s)$ provided that $a, a' < s(n)$ and $a \neq a'$.

$\square$

This means that an element of S_n is capable of holding $s(n)$ bits of information. The call $\mathit{read}(w, a, s)$ reads the a-th bit contained in s; the call $\mathit{write}(w, a, b, s)$ sets it to b when $a < s(n)$. Otherwise, the behaviour of these functions is left unspecified.

The first argument w plays the role of a “scratch pad”; its contents are unimportant; it is used as an item of auxiliary space to perform reading and writing. Both *read* and *write* return an equally long list for possible subsequent use as a scratch pad. Similarly, the address a and (in case of *read* the store s itself) are being returned as part of the result. In a linear setting this is crucial as otherwise these arguments would be lost.

Lemma 4.3 *Let $c \in \mathbb{N}$ be a constant. There is a $(0, \lambda n.c)$-storage device.*

Proof. For $n \in \mathbb{N}$ we put $S = S_n = T^c$

We put

$$\begin{aligned}
&\mathit{init} = (\mathtt{tt}, \dots, \mathtt{tt})\\
&\mathit{read}(w, a, s) = (w, a, b_a, s), \text{ if } a < c\\
&\mathit{read}(w, a, s) = (w, a, \mathtt{tt}, s), \text{ otherwise}\\
&\mathit{write}(w, a, b, s) = (w, a, (b_0, \dots, b_{a-1}, b, b_{a+1}, \dots, b_{c-1})), \text{ if } a < c\\
&\mathit{write}(w, a, b, s) = (w, a, s), \text{ otherwise}
\end{aligned}$$

when $s = (b_0, \dots, b_{c-1})$.

We have $S = [\![\mathsf{S}]\!]$ where $\mathsf{S} = \mathsf{T} \otimes \dots \otimes \mathsf{T})$ with c factors. Since c is a constant we can “hardwire” all possible c addresses, i.e., we use a case distinction on a of depth $\log(c)$ to distinguish all possible different values of a. We omit the details. $\square$

The key to larger sizes is the following lemma which shows how to "hide" information inside a (constant) function:

Lemma 4.4 *Let* S *be any type and put* $S = [\![\mathsf{S}]\!]$. *There is a representable functional*

$$\Phi : \mathsf{L}(S) \longrightarrow (W \to \mathsf{L}(S)) \times W \tag{9}$$

with the property

$$\Phi(l) = (f, w) \Rightarrow |w| = |l| \wedge \forall w'.|w'| = |l| \Rightarrow f(w') = l \tag{10}$$

Proof. The following program represents f:

```
$e_\Phi(l) = \mathsf{match}\ l\ \mathsf{with}$
  $\mathsf{nil} \Rightarrow (\lambda x.\mathsf{nil}) \otimes \mathsf{nil}$
  $\mathsf{cons}(d, s, l') \Rightarrow \mathsf{match}\ \Phi(l')\ \mathsf{with}$
    $f \otimes w \Rightarrow$
      $(\lambda x.\mathsf{match}\ x\ \mathsf{with}$
        $\mathsf{nil} \Rightarrow \mathsf{nil}$
        $\mathsf{cons}(d', b, w') \Rightarrow \mathsf{cons}(d', s, f(w')))$
      $\otimes \mathsf{cons}(d, \mathsf{tt}, w)$
```

□

The idea is that if $\Phi(l) = (f, w)$ then f holds all the information contained in l yet the abstract space (in the form of $\diamond$-values) occupied by l is returned as w. Of course, in order to read the information contained in f we need an argument of size $|l|$.

Lemma 4.5 *If there exists a* (k, s)*-storage device then there exists a* $(k+1, \lambda n.n \cdot s(n))$*-storage device.*

Proof. Suppose the storage device of size s is given by the sets $S_n \subseteq S$ and the functions *init*, *read*, *write*. We define the desired storage device on

$$S' = W \to \mathsf{L}(S)_\perp = [\![\mathsf{L}(\mathsf{T}) \multimap \mathsf{L}(\mathsf{S})]\!] \tag{11}$$

where $[\![\mathsf{S}]\!] = S$ and

$$S'_n = \{f \mid \forall w \in W_n.f(w) \in \mathsf{L}_n(S_n)\} \subseteq S' \tag{12}$$

We put

$$\mathit{init}'([]) = []$$
$$\mathit{init}'(x :: w) = \mathit{init}() :: \mathit{init}'(w)$$

so that $\mathit{init}' \in S'$.

Notice that we have $\mathit{init}' = [\![\mathtt{init'}]\!]$ where

$$e_{\mathtt{init'}} = \lambda w.\mathsf{match}\ w\ \mathsf{with}\ \mathsf{nil} \Rightarrow \mathsf{nil} \mid \mathsf{cons}(d, x, w_1) \Rightarrow \mathsf{cons}(d, \mathtt{init}(), \mathtt{init'}(w_1))$$

The definition of *read'* will be given as a sequence of intermediate results assuming the existence of certain helper functions whose definition we omit.

For *read'*(w, a, f) we start with $w, a \in W$ and $f \in S'$. We intend that $w \in W_{(k+1)n}$, $a \in W_n$, $f \in S'_n$ for some $n \in \mathbb{N}$.

We split w into w_1, w_2 such that $|w_1| + |w_2| = |w|$ and $|w_1|/|w_2| = 1/k$. If this is impossible we immediately produce some default result. Notice that if $|w| = (k+1)n$ as intended then such decomposition is possible and $|w_1| = |a| = n, |w_2| = kn$. We now apply f to w_1 yielding $l \in \mathsf{L}(S)$, actually $l \in \mathsf{L}_n(S_n)$ in case $f \in S'_n$. We decompose l into $l_1, l_2 \in \mathsf{L}(S), s \in S, d \in \diamond$ where $l_1 @ [s] @ l_2 = l$ and s is the $(a \bmod |a|)$-th entry of l. We let a_1 be $a \operatorname{div} |a|$ where $|a_1| = |a| = n$ and call $\mathrm{read}(w_2, a_1, s)$. This yields the desired boolean value b which forms the main result of *read'*(w_2, a, s). The other return values comprise s and a list w'_2 with $|w'_2| = kn = |w_2|$. From s, l_1, l_2, d we reconstruct l and then—using Lemma 4.4—we reconstruct f and obtain w'_1 with $|w'_1| = |w_1| = n$. We return $w'_1 @ w'_2, a_1, b, f$.

The definition of *write'* is analogous. □

Proof of Theorem 4.1 Suppose that $f : W \to W$ is a function such that $f(l)$ is computable on a Turing machine M in time $2^{p(|l|)}$. Let k be the degree of p. By Lemmas 4.3 and 4.5 there exists a $(k, \lambda n.p(2kn))$-storage device S.

This means that in the presence of a list $w \in W_{n/2}$ serving as a scratch pad we can store $p(n)$ bits.

Starting from the input presented as an element $l \in W$ where $n = |l|$ we first construct by recursion on l an element $(w, l') \in W_{n/2} \times \mathsf{L}_{n/2}(T \times T)$ such that l' contains the entire information of l. Notice that this is possible as a diagonal map $diag : T \to T \times T$ with $diag(x) = (x, x)$ is definable by $e_{\mathsf{diag}}(x) = \mathsf{if}\ x\ \mathsf{then}\ \mathsf{tt} \otimes \mathsf{tt}\ \mathsf{else}\ \mathsf{ff} \otimes \mathsf{ff}$. Alternatively, we can use rule CONTR.

Thus w can be used as a scratch pad for the storage device to store the required amount of $p(n)$ bits occurring as work tape inscriptions. Additionally we can simulate an unbounded stack by general recursion, see [7] for details.

Thus, by Cook's result [4] the function f is representable. □

We will now provide a corresponding upper bound on expressivity:

Theorem 4.6 *If $f : W \to W$ is representable then $f(l)$ is computable on a deterministic Turing machine in time $O(2^{p(|l|)})$ for some polynomial p.*

The proof of this result is based on two intuitions: Firstly, due to the linear typing discipline the size of all intermediate results is a priori bounded by a function of the size of the input. Second, linear functions can be simulated as argument-result pairs if one allows for nondeterminism: when constructing a linear function one guesses an argument and stores it together with the corresponding result. When applying such a linear function, one checks whether the actual argument agrees with the previously guessed one and in this case returns the precomputed result. Otherwise, the result is undefined.

To make this precise we construct an appropriate finite relational model for the language and show that evaluation in that finite model yields the same result as evaluation in the official order-theoretic (infinite) model.

Let $N \in \mathbb{N}$ be a fixed parameter. We define finite sets $(\![A]\!)$ together with functions $|-|_A : (\![A]\!) \to \{0,\dots,N\}$ for types A inductively as follows.

$$
\begin{array}{ll}
(\![\Diamond]\!) = \{0\} & |0|_\Diamond = 1 \\
(\![\mathsf{T}]\!) = \{\mathtt{tt}, \mathtt{ff}\} & |x|_\mathsf{T} = 0 \\
(\![\mathsf{L}(A)]\!) = \{w \in \mathsf{L}((\![A]\!)) \mid |w|_{\mathsf{L}(A)} \leq N\} & |[a_1,\dots,a_n]|_{\mathsf{L}(A)} = n + \sum_{i=1}^n |a_i|_A \\
(\![A \otimes B]\!) = \{x \in (\![A]\!) \times (\![B]\!) \mid |x|_{A\otimes B} \leq N\} & |(a,b)|_{A\otimes B} = |a|_A + |b|_B \\
(\![A \multimap B]\!) = (\![A]\!) \times (\![B]\!) & |(a,b)|_{A\multimap B} = |b|_B \dot{-} |a|_A
\end{array}
$$

For context Γ we define

$$(\![\Gamma]\!) = \{\eta \mid \forall x \in \mathrm{dom}(\Gamma).\eta(x) \in (\![\Gamma(x)]\!) \wedge |\eta|_\Gamma \leq N\} \qquad |\eta|_\Gamma = \textstyle\sum_{x\in\mathrm{dom}(\Gamma)} |\eta(x)|_{\Gamma(x)}$$

When we use, e.g., $|x|_{A\otimes B}$ in the definition of $(\![A \otimes B]\!)$ it refers to the defining expression for $|-|_{A\otimes B}$ given afterwards. The "modified difference" $x \dot{-} y$ is defined as $x - y$ if $x > y$ and 0 otherwise. Notice that for nonnegative numbers x, y, z one has $x + y \geq z$ iff $x \geq z \dot{-} y$.

For $U \subseteq (\![A]\!)$ we define $|U|_A = \max_{a\in U} |a|_A$.

A *relational valuation* of a signature Σ assigns to each $f : (A_1,\dots,A_r) \to B$ declared in Σ a relation

$$\rho(f) \subseteq (\![A_1 \otimes \dots A_r]\!)^n \times (\![B]\!) \tag{13}$$

such that $(a_1,\dots,a_n)\rho(f)b$ implies $|b|_B \leq \sum_{i=1}^n |a_i|_{A_i}$.

Given relational valuation ρ of Σ we define a relation

$$(\![e]\!)_\rho \subseteq (\![\Gamma]\!) \times (\![A]\!) \tag{14}$$

by induction on a typing derivation $\Gamma \vdash_\Sigma e : A$ as follows:

$$
\begin{array}{lr}
(\!|\Gamma \vdash x : \Gamma(x)|\!)_\rho = \{(\eta, v) \mid v = \eta(x)\} & (\textsc{Var}) \\
(\!|\Gamma_1, \ldots, \Gamma_r \vdash f(e_1, \ldots, e_r) : B|\!)_\rho = \{(\eta, v) \mid & \\
\quad \eta = \eta_1 \uplus \cdots \uplus \eta_r \wedge & \\
\quad \bigwedge_i \eta_i (\!|\Gamma_i \vdash e_i : A_i|\!)_\rho v_i \wedge (v_1, \ldots, v_r)\rho(f)v\} & (\textsc{Sig}) \\
(\!|\Gamma, x{:}A \vdash e : B|\!)_\rho = \{(\eta, v) \mid & \\
\quad \eta \uplus [y \mapsto \eta(x)](\!|\Gamma, x{:}A, y{:}A \vdash e : B|\!)_\rho v\} & (\textsc{Contr}) \\
(\!|\Gamma \vdash c : \mathsf{T}|\!)_\rho = \{(\eta, v) \mid \eta \in (\!|\Gamma|\!), v = [\![c]\!]\} & (\textsc{Const}) \\
(\!|\Gamma, \Delta \vdash \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A|\!)_\rho = \{(\eta, v) \mid & \\
\quad \eta = \eta_1 \uplus \eta_2 \wedge (& \\
\quad\quad \eta_1 (\!|\Gamma \vdash e : \mathsf{T}|\!)_\rho \mathsf{tt} \wedge \eta_2 (\!|\Delta \vdash e' : A|\!)_\rho v \vee & \\
\quad\quad \eta_1 (\!|\Gamma \vdash e : \mathsf{T}|\!)_\rho \mathsf{ff} \wedge \eta_2 (\!|\Delta \vdash e'' : A|\!)_\rho v)\} & (\textsc{If}) \\
(\!|\Gamma \vdash \mathsf{nil} : \mathsf{L}(A)|\!)_\rho = \{(\eta, []) \mid \eta \in (\!|\Gamma|\!) & (\textsc{Nil}) \\
(\!|\Gamma_d, \Gamma_h, \Gamma_t \vdash \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)|\!)^n_\rho = \{(\eta, v_h :: v_t) \mid & \\
\quad \eta = \eta_d \uplus \eta_h \uplus \eta_t \wedge & \\
\quad \eta_d (\!|\Gamma_d \vdash e_d : \diamond|\!)_\rho 0 \wedge & \\
\quad \eta_h (\!|\Gamma_h \vdash e_h : A|\!)_\rho v_h \wedge & \\
\quad \eta_t (\!|\Gamma_t \vdash e_t : \mathsf{L}(A)|\!)_\rho v_t\} & (\textsc{Cons}) \\
(\!|\Gamma, \Delta \vdash \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} \mid \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}} : B|\!)^n_\rho = \{(\eta, v) \mid & \\
\quad \eta = \eta_1 \uplus \eta_2 & \\
\quad \eta_1 (\!|\Gamma \vdash e : \mathsf{L}(A)|\!)_\rho [] \wedge \eta_2 (\!|\Delta \vdash e_{\mathsf{nil}} : B|\!)_\rho v \vee & \\
\quad \eta_1 (\!|\Gamma \vdash e : \mathsf{L}(A)|\!)_\rho v_h :: v_t \wedge & \\
\quad\quad \eta_2 [d \mapsto 0, h \mapsto v_h, t \mapsto v_t](\!|\Delta, d{:}\diamond, h{:}A, t{:}\mathsf{L}(A) \vdash e_{\mathsf{cons}} : A|\!)_\rho v\} & (\textsc{List-Elim}) \\
(\!|\Gamma, \Delta \vdash e_1 \otimes e_2 : A \otimes B|\!)^n_\rho = \{(\eta, (v_1, v_2)) \mid & \\
\quad \eta_1 (\!|\Gamma \vdash e_1 : A|\!)_\rho v_1 \wedge \eta_2 (\!|\Delta \vdash e_2 : B|\!)_\rho v_2\} & (\textsc{Pair}) \\
(\!|\Gamma, \Delta \vdash \mathsf{match}\ e_1\ \mathsf{with}\ x \otimes y \Rightarrow e_2 : C|\!)_\rho = \{(\eta, v) \mid & \\
\quad \eta = \eta_1 \uplus \eta_2 \wedge & \\
\quad \eta_1 (\!|\Gamma \vdash e_1 : A \otimes B|\!)_\rho (v_1, v_2) \wedge & \\
\quad \eta_2 [x \mapsto v_1, y \mapsto v_2](\!|\Delta, x{:}A, y{:}B \vdash e_2 : C|\!)_\rho v\} & (\textsc{Split}) \\
(\!|\Gamma \vdash \lambda x.e : A \multimap B|\!)_\rho = \{(\eta, (a, b)) \mid & \\
\quad \eta[x \mapsto a](\!|\Gamma, x{:}A \vdash e : B|\!)_\rho b\} & (\textsc{Lam}) \\
(\!|\Gamma, \Delta \vdash e_1 e_2 : B|\!)_\rho = \{(\eta, b) \mid & \\
\quad \eta = \eta_1 \uplus \eta_2 & \\
\quad \eta_1 (\!|\Gamma \vdash e_1 : A \multimap B|\!)(a, b) \wedge & \\
\quad \eta_2 (\!|\Delta \vdash e_2 : A|\!) a\} & (\textsc{App})
\end{array}
$$

The thus defined interpretation of a program is non size-increasing in the following sense.

Lemma 4.7 *If ρ is a relational valuation of Σ and $\Gamma \vdash_\Sigma e : A$ then whenever $\eta(\!|\Gamma \vdash e : A|\!)_\rho a$ one has $|a|_A \leq |\eta|_\Gamma$.*

Proof. Direct induction on typing derivations. □

For a given program P the mapping which sends ρ to the relational valuation

$$f \mapsto (\!|x_1{:}A_1, \ldots, x_n{:}A_n \vdash e_f : B|\!)_\rho \tag{15}$$

is clearly monotone (with respect to inclusion) so that we can define the relational semantics of a program as the least fixpoint of this functional which in view of the finiteness of the domains is actually reached after a finite number of iterations starting from the relational valuation assigning the empty relation to each function symbol.

We write $(\!|P|\!)(f)$ or simply $(\!|f|\!)$ for the thus obtained interpretation of a function symbol f in some program P. Since the empty relation is a relational valuation and by the previous lemma the semantics maps relational valuations to relational valuations, the thus defined semantics of a program is also a relational valuation, i.e., non size-increasing.

Proposition 4.8 *Suppose that P is a program containing some function symbol $f : (W) \to W$ and let $l \in W$ where $|l| \leq N$ (recall that N is a fixed parameter). Notice that in this case $l \in [\![\mathsf{L}(T)]\!]$ as well as $l \in (\!|\mathsf{L}(T)|\!)$. Then $l(\!|f|\!)l' \iff [\![f]\!](l) = l'$ for all $l' \in W$.*

This means in particular that $(\!|f|\!)$ is a partial function.

Before we prove this result let us remark that it allows us to evaluate any function $f : (\mathsf{L}(\mathsf{T})) \to \mathsf{L}(\mathsf{T})$ in a finite amount of time (regardless of its termination behaviour under an evaluation strategy based on rewriting) by computing $(\!|f|\!)$ for appropriate parameter N. We will later estimate the amount of time required for this so as to obtain the desired characterisation. Let us first come to the proof of the proposition, though:

Proof. For each $n \leq N$ we define inductively a family of simulation relations

$$\sim^n_A \subseteq [\![A]\!] \times \{U \subseteq (\!|A|\!) \mid U \neq \emptyset \wedge |U|_A \leq n\} \tag{16}$$

between elements of $[\![A]\!]$ and nonempty subsets of $(\!|A|\!)$ of size $\leq n$. Recall that $|U|_A = \max_{x \in U} |x|_A$.

To simplify the notation we introduce the following shorthands: if $U \subseteq (\!|A|\!)$ and $V \subseteq (\!|B|\!)$ then $U \times V := \{(a, b) \mid a \in U \wedge b \in V\}$ We have $U \times V \subseteq (\!|A \otimes B|\!)$ iff $|U|_A + |V|_B \leq N$ and in this case $|U \times V|_{A \otimes B} = |U|_A + |V|_B$.

If $U \subseteq (\!|A|\!)$ and $V \subseteq (\!|\mathsf{L}(A)|\!)$ then $U{::}V := \{a :: w \mid a \in U \wedge w \in V\}$ We have $U{::}V \subseteq (\!|\mathsf{L}(A)|\!)$ iff $|U|_A + |V|_{\mathsf{L}(A)} + 1 \leq N$ and in this case $|U :: V|_{\mathsf{L}(A)} = |U|_A + |V|_B + 1$.

If $U \subseteq (\!|A \multimap B|\!)$ and $V \subseteq (\!|A|\!)$ then $U(V) := \{b \mid \exists a \in V.(a, b) \in U\}$ We have $|U(V)|_B \leq |U|_{A \multimap B} + |V|_A$.

We formally extend $\sim^n_A$ by putting $\perp \sim^n_A \emptyset$. Notice that whenever $x \in [\![A]\!] \cup \{\perp\}$ and $x \sim^n_A U$ and $x \neq \perp$ then $U \neq \emptyset$.

The defining clauses are now given as follows.

$$\mathsf{tt} \sim^n_{\mathsf{T}} \{\mathsf{tt}\} \quad \mathsf{ff} \sim^n_{\mathsf{T}} \{\mathsf{ff}\} \quad 0 \sim^{n+1}_{\diamond} \{0\} \quad [] \sim^n_{\mathsf{L}(A)} \{[]\}$$

$$\begin{array}{lcl}
(a, b) \sim^n_{A \otimes B} W & \iff & \exists n_1, n_2, U, V. n_1 + n_2 = n \\
 & & \quad \wedge\ a \sim^{n_1}_A U \wedge b \sim^{n_2}_B V \wedge W = U \times V \\
f \sim^n_{A \multimap B} U & \iff & \forall n_1, x. n + n_1 \leq N \\
 & & \quad \wedge\ x \sim^{n_1} V \Rightarrow f(x) \sim^{n+n_1}_B U(V) \\
x :: l \sim^n_{\mathsf{L}(A)} W & \iff & \exists n_1, n_2, U, V. n_1 + n_2 + 1 \leq n \\
 & & \quad \wedge\ x \sim^{n_1}_A U \wedge l \sim^{n_2}_{\mathsf{L}(A)} V \wedge W = U :: V
\end{array}$$

Notice that if $m \leq n \leq N$ then $x \sim^m_A U$ implies $x \sim^n_A U$. Notice also that if A is heap-free and $x \sim_A U$ then U has at most one element; exactly one if $x \neq \perp$. We write $\eta \sim^n_\Gamma U$ for $\eta \in [\![\Gamma]\!]$ and $U \subseteq (\!|\Gamma|\!)$ if there exist $\mathrm{dom}(\Gamma)$-indexed families $(n_x)_x, (U_x)_x$ such that $\sum_{x \in \mathrm{dom}(\Gamma)} n_x \leq n$

and $U = \prod_{x \in \mathrm{dom}(\Gamma)} U_x$ and $\eta(x) \sim^{n_x}_{\Gamma(x)} U_x$ for all $x \in \mathrm{dom}(\Gamma)$. If X, Y are sets and $f : X \to Y$ and $U \subseteq X$ we define

$$f(U) := \{y \in Y \mid \exists x \in U. y \in f(x)\} \tag{17}$$

Similarly, if $\Gamma \vdash e : A$ and $U \subseteq (\!|\Gamma|\!)$ we define

$$(\!|\Gamma \vdash e : A|\!)_{U,\rho} = \{b \mid \exists \eta \in U. \eta (\!|\Gamma \vdash e : A|\!)_{\rho} b\} \tag{18}$$

Suppose that we are given a domain-theoretic valuation ψ and a relational valuation ρ of a given signature Σ. We will write $\psi \sim \rho$ to mean that for each function symbol $f : (A_1, \dots, A_r) \to B$ declared in Σ and whenever $n = n_1 + \cdots + n_r \leq N$ one has $\bigwedge_i u_i \sim^{n_i}_{A_i} U_i \Longrightarrow \psi(f)(u_1, \dots, u_r) \sim^n_B \rho(f)^n(U_1, \dots, U_r)$ We now have the following sublemma :

Sublemma: *Suppose that* $\psi \sim \rho$. *If* $\Gamma \vdash_\Sigma e : A$ *and* $\eta \sim^n_\Gamma U$ *then* $[\![e]\!]_{\eta,\psi} \sim^n_A (\!|\Gamma \vdash e : A|\!)_{U,\rho}$

Proof of sublemma: By induction on typing derivations. For rule VAR we use the fact that U is nonempty.

Rule SIG follows from the assumption made on ψ and ρ.

Rule CONTR uses the fact that elements of heap-free type have zero size as well as the observation that whenever $v \sim_A U$ for heap-free A then U has at most one element which implies that whenever $\eta \sim_{\Gamma,x:A,y:A} U$ where $U_x = U_y$ and $\eta \in U$ then $\eta = \eta[y \mapsto \eta(x)]$. These are the only two properties of heap-free types used thus allowing for possible extensions. All other cases are direct.

□

Now let ψ_0 be the valuation defined by $\psi_0(f)(\vec{x}) = \bot$ and ρ_0 be the relational valuation that assigns the empty relation to each function symbol. Clearly, $\psi_0 \sim \rho_0$ and so the sublemma shows that $\psi_m \sim \rho_m$ for all m where

$$\begin{aligned} \psi_{m+1}(f)(v_1, \dots, v_r) &= [\![e_f]\!]_{[x_1 \mapsto v_1, \dots, x_r \mapsto v_r], \psi_m} \\ \rho_{m+1}(f)(v_1, \dots, v_r) &= (\!|e_f|\!)_{[x_1 \mapsto v_1, \dots, x_r \mapsto v_r], \rho_m} \end{aligned} \tag{19}$$

As already mentioned, in view of the finiteness of the sets $(\!|A|\!)$ there exists m_0 such that $(\!|P|\!)(f) = \rho_{m_0}(f)$ for all $f \in \mathrm{dom}(\Sigma)$. Therefore, $\forall m \geq m_0. \rho_m \sim (\!|P|\!)$.

Now, $[\![P]\!](f) = \bigvee_m \rho_m(f) = \bigvee_{m \geq m_0} \rho_m(f)$. It is readily seen by induction on types that each relation $\sim^n_A$ is continuous in the sense that $\forall i. x_i \sim^n_A U$ implies $(\bigvee_i x_i) \sim^n_A U$ assuming of course that the x_i form an ascending chain. We have thus proved that $[\![P]\!] \sim (\!|P|\!)$ which yields the desired result when specialised to the type $\mathsf{L}(\mathsf{T})$. □

The idea is now to compute for a given N the iterations ρ_m by stepwise updating a big value table holding the relations $\rho_m(f)$.

To estimate the size of such a value table we must estimate the number of elements of the sets $(\!|A|\!)$. Writing $\#X$ for the cardinality of set X we have

$$\begin{array}{lll} \log \#(\!|\mathsf{T}|\!) = 1 & \log \#(\!|\Diamond|\!) = 0 & \log \#(\!|\mathsf{L}(A)|\!) \leq N \log \#(\!|A|\!) \\ \log \#(\!|A \otimes B|\!) \leq \log \#(\!|A|\!) + \log \#(\!|B|\!) & & \log \#(\!|A \multimap B|\!) \leq \log \#(\!|A|\!) + \log \#(\!|B|\!) \end{array} \tag{20}$$

Therefore, for a given program P we can find a polynomial p such that $\log \#(\!|A|\!) \leq p(N)$ for each type A occurring in P.

The space required to store a relational valuation for P in the relational model is therefore $O(2^{p(N)})$ where the hidden constant involves the number and arities of function symbols.

Now, using the definition of $(\!|\Gamma \vdash e : A|\!)$ the computation of ρ_{m+1} given a value table for ρ_m and space to hold ρ_{m+1} can be performed with $O(p(N))$ extra space which would be required e.g. to hold particular elements of $(\!|A|\!)$.

In order to compute $(\!|P|\!)$ we maintain space for two value tables initialising both with the empty relational valuation. If at any time one of the two tables holds ρ_m we perform the necessary computations to achieve that the other one holds ρ_{m+1}. Thereafter, ρ_m is not needed anymore so that we can overwrite it with ρ_{m+2} and so forth, until no more changes take place and we have found $(\!|P|\!)$.

Since $\rho_m \subseteq \rho_{m+1}$ the number of iterations is $O(2^{p(N)})$ as well (in the worst case each iteration adds one single tuple to ρ), so that we have given a $DTIME(O(2^{p(N)}))$ algorithm for computing $(\!|P|\!)$ hence $[\![P]\!](f)(l)$ for $f : (\mathsf{L}(\mathsf{T}))\to\mathsf{L}(\mathsf{T})$ when $|l| \leq N$.

References

[1] Klaus Aehlig and Helmut Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. In *Proceedings of the Fifteenth IEEE Symposium on Logic in Computer Science (LICS '00), Santa Barbara*, 2000.

[2] Stephen Bellantoni and Stephen Cook. New recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2:97–110, 1992.

[3] Vuokko-Helena Caseiro. *Equations for Defining Poly-time Functions.* PhD thesis, University of Oslo, 1997. Available by ftp from `ftp.ifi.uio.no/pub/vuokko/0adm.ps`.

[4] Stephen A. Cook. Linear-time simulation of deterministic two-way pushdown automata. *Information Processing*, 71:75–80, 1972.

[5] Andreas Goerdt. Characterizing complexity classes by higher type primitive recursive definitions. *Theoretical Computer Science*, 100:45–66, 1992.

[6] Martin Hofmann. Linear types and non size-increasing polynomial time computation. To appear in Theoretical Computer Science. See `www.dcs.ed.ac.uk/home/papers/icc.ps.gz` for a draft. An extended abstract has appeared under the same title in Proc. Symp. Logic in Comp. Sci. (LICS) 1999, Trento, 2000.

[7] Martin Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 2001. To appear, see `www.dcs.ed.ac.uk/home/mxh/papers/nordic.ps.gz` for a draft. An extended abstract has appeared in *Programming Languages and Systems*, G. Smolka, ed., Springer LNCS, 2000.

[8] Neil Jones. The Expressive Power of Higher-Order Types or, Life without CONS. *Journal of Functional Programming*, 2001. to appear.

[9] Daniel Leivant. Stratified Functional Programs and Computational Complexity. In *Proc. 20th IEEE Symp. on Principles of Programming Languages*, 1993.

[10] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993.

Program Analysis for Implicit Computational Complexity

Neil D. Jones
DIKU, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
e-mail: neil@diku.dk

This talk brings together ideas from two lines: *automatic estimation of program running times*, and *implicit computational complexity*. It describes ongoing research. Recent work in the two areas has been done by Bellantoni and Cook, Benzinger, Hofmann, Jones, Crary and Weirich, Leivant, Marion, Schwichtenberg, and others.

A main goal of implicit computational complexity is to "capture" complexity classes such as PTIMEF (polynomial-time computable functions) by computing formalisms that do not impose explicit bounds on time or space resources. Several researchers have succeeded in reaching this goal, a well-known example being the Bellantoni-Cook "safe primitive recursion on notation."

It must be said, however, that recursion-theoretic formalisms such as primitive recursion are not very close to programming practice. In particular natural *algorithms*, as seen in introductory algorithm courses, often do not fall into existing implicit complexity classes. In some cases this has even been proven impossible, e.g., Colson established that primitive recursion alone cannot express computing the minimum of two numbers by the obvious linear-time algorithm.

In this work we identify a decidable class of algorithms such that all can be executed within polynomial time (or logarithmic space); and as well, the class includes many natural algorithms that are used in solving real problems.

For a standard first-order functional language we devise a type system giving information on the *variations* of its function parameters in terms of program inputs, and on *run-time bounds* for program-defined functions. Every syntactically correct program is well-typed, i.e., the language has a so-called "soft" type system.

The type information is extracted by data-flow analysis algorithms that extend the "size-change" framework of our POPL 2001 paper to account for running times as well as termination. The analysis allows automatic detection of programs that are guaranteed to run (or be runnable) in polynomial time.

Theorems are proven that this is indeed the case; and that the class is a proper generalization of "safe recursion" and some related schemes provided by other researchers. Several representative natural and efficient algorithms are seen to fall into the class, providing evidence that the class is "large enough."

Proof mining in functional analysis

Ulrich Kohlenbach

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark
kohlenb@brics.dk
http://www.brics.dk/~kohlenb/

By the term 'proof mining' we denote the activity of transforming a prima facie non-constructive proof into a new one from which certain computational information can be read off which was not visible beforehand. That new proof in general will not and need not be fully constructive. On the contrary, usually only small parts of a given proof need to be considered at all. Proof-theoretic techniques based on a monotone version of functional interpretation and additional specially designed proof interpretations allow such extractions of effective data from fairly large classes of proofs ([3],[4]). Moreover, these proof interpretations show which parts of a given proof are relevant for its computational content.

The area of analysis (and in particular numerical functional analysis) is of special interest in connection with proof mining since here non-effectivity is due not only to the use of non-constructive logical reasoning but at the core of many principles (like compactness arguments) which are used to ensure convergence and which provably rely on the existence of non-computable reals. In mathematical terms this non-computability often is an obstacle to obtain a quantitative stability analysis and rates of convergence.

We report on the results of a recent case study of this proof-theoretic approach to computability and complexity in analysis in the fixed point theory of non-expansive mappings, one of the most active areas in nonlinear functional analysis.

Our results provide even new qualitative information about the asymptotic regularity of so-called Krasnoselski-Mann iterations and strengthen classical theorems of Ishikawa ([2]) and Borwein-Reich-Shafrir ([1]).

References

[1] Borwein, J., Reich, S., Shafrir, I., Krasnoselski-Mann iterations in normed spaces. Canad. Math. Bull. **35**, pp. 21-28 (1992).

[2] Ishikawa, S., Fixed points and iterations of a nonexpansive mapping in a Banach space. Proc. Amer. Math. Soc. **59**, pp. 65-71 (1976).

[3] Kohlenbach, U., Analysing proofs in analysis. In: W. Hodges, M. Hyland, C. Steinhorn, J. Truss, editors, *Logic: from Foundations to Applications. European Logic Colloquium* (Keele, 1993), pp. 225–260, Oxford University Press (1996).

[4] Kohlenbach, U., Arithmetizing proofs in analysis. In: Larrazabal, J.M. et al. (eds.), Proceedings Logic Colloquium 96 (San Sebastian), Springer Lecture Notes in Logic **12**, pp. 115–158 (1998).

[5] Kohlenbach, U., On the computational content of the Krasnoselski and Ishikawa fixed point theorems. To appear in: Computability and Complexity in Analysis (CCA 2000), J. Blanck, V. Brattka, P. Hertling, K. Weihrauch (eds.), Springer LNCS 2064 (2001).

[6] Kohlenbach, U., A quantitative version of a theorem due to Borwein-Reich-Shafrir. To appear in: Numerical Functional Analysis and Optimization.

TERMINATION PROOFS AND COMPLEXITY CERTIFICATION

Extended Abstract

March 2001

Daniel Leivant*

Abstract

We show that simple structural conditions on proofs of convergence of functional programs, in the intrinsic-theories verification framework of [20], correspond to resource bounds on program execution. These conditions may be construed as reflecting finitistic-predicative reasoning. The results provide a user-transparent method for certifying the computational complexity of functional programs. In particular, we define natural notions of data-predicative formulas and of data-predicative derivations, and show that induction for data-predicative formulas captures precisely the primitive recursive functions, data-predicative derivations the Kalmar-elementary functions, and the combination of both the poly-time functions.

1 INTRODUCTION

1.1 Main results

In [20] we put forth, for each sorted inductive algebra $\mathbb{A}(C)$ over a set C of generators, a formalism $\mathbf{IT}(C)$ for reasoning about recursive equational programs over $\mathbb{A}(C)$. We showed that the resulting formalisms have the same proof theoretic power as first order (Peano) arithmetic. Here we consider simple structural properties on formulas in derivations, which guarantee that a program proved terminating in fact runs within certain resource bounds.

We consider two forms of structural restrictions on natural deductions:

1. Induction is restricted to "data-predicative" formulas; these are the formulas where data-predicates do not occur both positively and negatively; they properly include the $\mathbf{\Sigma}^0_1$ and $\mathbf{\Pi}^0_1$ formulas of **PA**.

2. There are no data-positive working assumptions, i.e. assumptions that are closed in the derivation, and where data-predicates occur positively. It suffice to disallow such assumptions over the major (i.e. leftmost) premise of induction.

We show that programs whose termination is provable under these restrictions characterize exactly major complexity classes, as listed below: every provable program uses the indicated resources, and every function in the indicated complexity class is computable by some program provable with the indicated restrictions:

- Condition (1) yields exactly the primitive recursive functions (Theorem 3).

*Computer Science Department, Indiana University, Bloomington, IN 47405. `leivant@cs.indiana.edu`. Research partially supported by NSF grant DMS-9870320

- (2) yields exactly the (Kalmar-) elementary functions (Theorem 4(1)).
- (1) and (2) combined yield linear space for programs over $\mathbb{N}$, and poly-time for programs over symbolic data, e.g. $\{0,1\}^*$ (Theorem 5(2)).

Restrictions (1) and (2) can be construed as reflecting a certain "finitistic-predicative" view of data. In broad strokes, the underlying views can be stated as follows. Take the position that an inductive data system, say the set $\mathbb{N}$ of natural numbers, comes into being through a process, but cannot be admitted as a completed totality. In particular, determining the elements of $\mathbb{N}$ should not depend on assuming that the full outcome of the generative process for $\mathbb{N}$ is already in hand. To constructively justify an inference by Induction of a formula $\varphi[\mathbf{t}]$ one needs then to posit that the induction formula $\varphi[x]$ is well-defined for all values for x, and that the induction eigen-term $\mathbf{t}$ is well-defined. A formula in which $\mathbf{N}$ occurs only positively, say $\exists x\, \mathbf{N}(f(x))$, asserts that the process of generating $\mathbb{N}$ eeventually yields an element in the range of f; no invocation of $\mathbb{N}$ as completed totality is needed. This justifies induction for data-non-negative formulas. For data-non-positive formulas, the rationale is sensitive to the underlying logic: with the principle of excluded third available, data-non-positive and data-non-negative induction are equivalent. However, in constructive logic, data-non-positive formulas are extremely weak; roughly, knowing that an object x is *not* obtained by an inductive process yields no computationally useful information.

Consider, in contrast, a non-data-predicative formula, say $\forall x\, \mathbf{N}(x) \rightarrow \mathbf{N}(\mathbf{f}(x))$, which classically is equivalent to $\forall x\, \neg\mathbf{N}(x) \vee \mathbf{N}(\mathbf{f}(x))$. The meaning of such a formula clearly depends on admitting $\mathbb{N}$ as a completed totality.

The consequence of the finitistic-predicative viewpoint is even more dramatic when it comes to the eigen-term $\mathbf{t}$ of induction (recall that in our framework $\mathbf{t}$ may contain arbitrary function identifiers). Consider our Data-Elimination (i.e. Induction) rule, say for $\mathbb{N}$,

$$\frac{\mathbf{N}(\mathbf{t}) \quad \varphi[\mathbf{0}] \quad \cdots}{\varphi[\mathbf{t}]}$$

If we assert the major premise, $\mathbf{N}(\mathbf{t}[x])$, on the basis of $\mathbf{N}(x)$ as assumption, and x is later quantified in the derivation, then we implicitly posit that the scope of $\mathbb{N}$ is well delineated.

1.2 Motivation and benefits

The simple framework of [20] yields surprisingly rich benefits. Practically, it enables explicit reasoning about functional programs without recourse to numeric codes or a logic of partially-denoting terms. This includes programs whose termination cannot be proved within the formal theory used. Conceptually, the framework lends itself to a delineation of various forms of finitistic and predicative ontologies of data, and to proof theoretic characterizations of computational complexity classes, as we do here. Such formalisms are quite different from the well developed framework of Bounded Arithmetic, and offer an expressively rich and unobtrusive setting for formalizing Feasible Mathematics, e.g. Poly-time or Poly-space Mathematics.

This novel proof theoretic framework has other promising applications. For instance, it yields a simple and attractive notion of provable functions of all finite types, with, as a result, natural notions of feasibility in higher types. In pure proof theory, intrinsic theories lead to natural definitions of a constructive analogue of Kleene's arithmetical hierarchy. (The hierarchy is infinite, but relations defined by prenex formulas are at level 2.) These and related issues will be presented in other papers, and are mentioned here to illustrate the scope of the method.

1.3 Comparisons

Several connections have been discovered between proof complexity and the computational complexity of provably-terminating programs. Gödel's famed Dialectica translation [5] can already be seen as such a connection, showing that the provably recursive functions of first-order arithmetic are precisely the functions definable by primitive-recursion in all finite types.[1]

More recent connections between the complexity of program-termination proofs and computational complexity have used a number of paradigms.

1. RESTRICTION ON INDUCTION FORMULAS. Parson [21] showed that restricting induction to $\mathbf{\Sigma}^0_1$ formulas yields precisely the primitive recursive functions, a special case of our Theorem 3 below. Inspired by Cobham's characterization of poly-time by bounded recursion, induction on bounded formulas was studied by Buss and others, leading to characterizations of several classes, notable poly-time and poly-space [4].

2. RESTRICTED SET EXISTENCE in second and higher order logic [16].

3. DATA RAMIFICATION. Data ramification for functional programming was introduced independently in [26, 1, 15], and was subsequently shown to yield functional languages that characterize precisely major computational complexity classes.

   First order theories based on ramification were introduced in [18].[2] The ramified theory with first order induction yields linear space for numeric data, and poly-space for symbolic data.

4. LINEAR RAMIFIED THEORIES. Data ramification for functional programs was reformulated by Hofmann as a type system based on linear logic, notably leading to a type system that allows recursion in all types without leading out of poly-time [8, 10, 9, 3]. A proof theoretic analogue of these formalisms were developed in [25, 2]. One the characterizations we give below for poly-time is also based on structural proof theoretic conditions akin to linear logic (they are based on ideas developed in [19], predating the papers above and independent of them.) However, our formalism is dramatically simpler than the ones above. In particular, they do no involve an extension of first order formulas with modal and linear operators. Even if it can be shown that more algorithms can be proved terminating in the formalisms of [25, 2], the availability of a much simpler framework with similar properties, is of fundamental interest.

Like the ramified formalisms mentioned under (3) above, the research reported here is based on the framework of intrinsic theories for inductive data [18, 20]. The salient aspects of this approach are (1) programs are referred to explicitly at the assertion level, not through coding or modal operations; and (2) there are explicit predicates to convey that terms denote data of particular sorts. However, contrary to (3) the restriction on proofs considered here are articulated in terms of restrictions on formulas in crucial positions in a proof, rather than a type-like labeling system. This approach offers a number of advantages, including a generic framework for direct and flexible reasoning about declarative programs (including divergent terms), a novel proof theoretic

[1]A new proof of this result, in the spirit of the project reported here, can be found in [20].
[2]A more complete account of this development is in [12].

treatment of inductive data and of the type hierarchy over such data, and transparent ways of verifying resource bounds for program execution. Here we report progress on the latter.

1.4 Technical background

In [20] we put forth, for each sorted inductive algebra $\mathbb{A}(C)$, over a set C of generators, a formalism $\mathbf{IT}(C)$ for reasoning about recursive equational programs over $\mathbb{A}(C)$. We refer the reader to the definitions and discussion there. The simplest non-degenerated inductively generated algebra is the set $\mathbb{N} = \mathbb{A}(\mathsf{0}, \mathsf{s})$ of natural numbers, generated from the constructors $\mathsf{0}$ and s (zero and successor). The intrinsic theory $\mathbf{IT}(\mathbb{N})$ is the first order theory over the vocabulary consisting of $\mathsf{0}$, s, and a unary relational identifier N. The axioms are:

1 *Generative axioms:* $\mathsf{N}(\mathsf{0})$ and $\forall x.\, \mathsf{N}(x) \rightarrow \mathsf{N}(\mathsf{s}x)$;

2 The axiom schema of *Induction*: $A[\mathsf{0}] \;\wedge\; \forall x.\,(A[x] \rightarrow A[\mathsf{s}x]) \;\rightarrow\; \forall x.\,(\mathsf{N}(x) \rightarrow A[x])$.

We also consider an extension $\overline{\mathbf{IT}}(\mathbb{N})$ of $\mathbf{IT}(\mathbb{N})$, with separation axioms for the constructors of $\mathbb{N}$ (i.e. Peano's third and fourth axioms):

3 $\forall x.\, \neg\, \mathsf{s}(x) = \mathsf{0}$,

4 $\forall x, y.\; \mathsf{s}x = \mathsf{s}y \rightarrow x = y$.

We shall use Gentzen-style natural deduction calculi for these theories, with the usual inference rules for equality, and a rendition of the axioms by inference rules:

Data-introduction

$$\frac{}{\mathsf{N}(\mathsf{0})} \qquad \text{and} \qquad \frac{\mathsf{N}(t)}{\mathsf{N}(\mathsf{s}t)}$$

Data-Elimination (i.e., induction)

$$\frac{\mathsf{N}(t) \quad \varphi[\mathsf{0}] \quad \begin{matrix}\{\varphi[x]\}\\ \vdots \\ \varphi[\mathsf{s}x]\end{matrix}}{\varphi[t]}$$

A degenerate form of data-elimination is

$$\frac{\mathsf{N}(t) \quad \varphi[\mathsf{0}] \quad \varphi[\mathsf{s}x]}{\varphi[t]}$$

that is, proof by cases on the main constructor.

Separation

$$\frac{\mathsf{s}t = t}{\bot} \qquad \text{and} \qquad \frac{\mathsf{s}t = \mathsf{s}t'}{t = t'}$$

We shall use freely common concepts and terminology for natural deduction calculi, as presented e.g. in [28]. We call a derivation of $\overline{\mathbf{IT}}(\mathbb{N})$ *simply-normal* if it is induction-free and normal (in the sense of first order logic). If $\mathcal{D}$ is a natural deduction derivation, we write $|\mathcal{D}|$ for its height.

We say that an r-ary function f over $\mathbb{N}$ is *provable* in $\mathbf{IT}(\mathbb{N})$ if there is an equational program $(P, \mathbf{f})$ that computes f, and for which

$$\mathbf{N}(\vec{x}) \vdash \mathbf{N}(\mathbf{f}(\vec{x}))$$

where $\mathbf{N}(\vec{x})$ abbreviates the conjunction $\mathbf{N}(x_1) \wedge \cdots \wedge \mathbf{N}(x_r)$, and provability is in $\mathbf{IT}(\mathbb{N}) + \bar{P}$, where $\bar{P}$ is the universal closure of the conjunction of the equations in P. Function provability in $\overline{\mathbf{IT}}(\mathbb{N})$ is defined analogously.

Analogous theories are defined for any inductively generated data-system $\mathbb{A}(C)$, where C is the set of constructors. When referring to a single-sorted data-system, such as $\mathbb{N}$ or $\mathbb{W} = \{0,1\}^*$, we use $\mathbf{D}$ as a generic predicate identifier for data, in par with our use of $\mathbf{N}$ above. Thus, for example, we have a data introduction rule for each constructor $\mathbf{c}$: if $\underline{arity}(\mathbf{c}) = r$, then the rule is

$$\frac{\mathbf{D}(x_1) \quad \cdots \quad \mathbf{D}(x_r)}{\mathbf{D}(\mathbf{c}(x_1 \ldots x_r))}$$

We refer the reader to [20] for detail and examples. We write $\mathbf{IT}(C)$ for the intrinsic theory for $\mathbb{A}(C)$, and $\overline{\mathbf{IT}}(C)$ for the extension of that theory with separation axioms.

In [20] we showed that the functions provable in $\mathbf{IT}(\mathbb{N})$, as well as those provable in $\overline{\mathbf{IT}}(\mathbb{N})$, are precisely the provably recursive functions of Peano Arithmetic. This remains true if in place of $\mathbb{N}$ we take any non-trivial inductive data-system. The underlying logic can be either classical or constructive (i.e. intuitionistic).

2 STATEMENT OF THE RESULTS

2.1 Data-predicative induction and primitive recursion

As usual, we say that an occurrence of $\mathbf{D}$ in a formula φ is *positive* (respectively, *negative*) if it is in the negative scope of an even (respectively, odd) number of implications and negations. We call a formula φ in the vocabulary of $\mathbf{IT}(C)$

- *data-positive* if $\mathbf{D}$ has positive occurrences in φ,
- *data-non-positive* if $\mathbf{D}$ has no positive occurrences,
- *data-non-negative* if $\mathbf{D}$ has no negative occurrences, and
- *data-predicative* if it does not have both positive and negative occurrences, i.e. is either *data-non-positive* or *data-non-negative.*

We define $\mathbf{IT}^+(C)$ to be $\mathbf{IT}(C)$ with induction restricted to data-predicative formulas. $\overline{\mathbf{IT}}^+(C)$ is defined analogously.

For example, formulas of the form $\exists x \; (\; \mathbf{D}(x) \; \wedge \; E \;)$, E an equation of primitive recursive arithmetic, are data-non-negative, and so data-predicative. These are precisely the interpretations in $\mathbf{IT}(C)$ of Σ^0_1 formulas of primitive recursive arithmetic. Similarly, all $\mathbf{D}$-free formulas are data-predicative, as are all formulas of the form $\mathcal{Q}.\,(\,\boldsymbol{\alpha} \rightarrow \mathbf{D}(t)\,)$ where $\mathcal{Q}$ is a block of quantifiers and $\boldsymbol{\alpha}$ is $\mathbf{D}$-free; these formulas are not interpretations in $\mathbf{IT}(C)$ of formulas of first order arithmetic.

The interpretation in $\mathbf{IT}(\mathbb{N})$ of a Π^0_1 formula $\forall x. E$, E an equation, is $\forall x. (\mathbf{N}(x) \rightarrow E)$, which is data-non-positive, and so also data-predicative. However, the interpretation of a $\mathbf{\Pi}^0_2$-formula $\forall x\, \exists y\, E$ is $\forall x\, (\mathbf{N}(x) \rightarrow \exists y\, \mathbf{N}(y) \wedge E)$, which is not data-predicative.

Let $\mathbb{A}(C)$ be an inductive data-system. Our results for induction restricted to data-predicative formulas are as follows.

PROPOSITION **1** *Every function definable by simultaneous recurrence over $\mathbb{A}(C)$ is provable in $\mathbf{IT}(C)$, based on minimal logic, and with induction for conjunctions of atomic formulas.*

In particular, all primitive recursive functions are provable in $\mathbf{IT}(\mathbb{N})$ using induction for such formulas.

PROPOSITION **2** *Every function provable in $\mathbf{IT}(C)$, based on classical logic, and with induction for data-predicative formulas, is definable by simultaneous recurrence over $\mathbb{A}(C)$. In particular, all provable functions of $\mathbf{IT}(\mathbb{N})$ are primitive recursive.*

From Propositions 1 and 2 we conclude

THEOREM **3** *The provable functions of $\mathbf{IT}^+(\mathbb{N})$ (or $\overline{\mathbf{IT}}^+(\mathbb{N})$) are precisely the primitive recursive functions.*

More generally, if C is a data-system, then the constructively provable functions of $\mathbf{IT}^+(C)$ (or $\overline{\mathbf{IT}}^+(C)$) are precisely the functions over the algebra $\mathbb{A}(C)$ that are generated from the constructors by composition and simultaneous recurrence. (These are also the functions that are primitive recursive modulo a canonical coding of C in $\mathbb{N}$.)

2.2 Data-predicative derivations

The results stated above show that the restriction of Induction to data-predicative formulas reduces the class of provable functions quite dramatically, from the provably recursive functions of first order arithmetic to the primitive recursive functions. However, from the viewpoint of feasible computation and computer science, the latter is still a stratospherically large class. In [18] we defined a ramified intrinsic theory of data, yielding provable functions that fall precisely into major complexity classes, notable the poly-time, linear space, and Kalmar-elementary functions, depending on allowable instances of induction and on the underlying data system. Here we use an alternative approach, where impredicative references to data in derivations are prohibited explicitly.

Call a natural deduction derivation $\mathcal{D}$ of $\mathbf{IT}(C)$ *data-predicative* if no major premise of non-degenerate data-elimination (i.e. induction) depends on a data-positive assumption closed in $\mathcal{D}$. As discussed above, this property reflects a finitistic-predicative concern.

We have:

THEOREM **4** *For any data-system $\mathbb{A}(C)$, a program P is provable in $\mathbf{IT}(C)$ by a data-predicative derivation iff P is computable in Kalmar-elementary resources.*

For the next result we refer to the generic register machines over a data-system $\mathbb{A}(C)$, as defined in [17].

THEOREM **5** *For any data-system $\mathbb{A}(C)$, a program P is provable in $\mathbf{IT}(C)$ by a data-predicative derivation with data-positive induction iff P is computable in poly-time on a register machine over $\mathbb{A}(C)$.*

In particular, P is provable in $\mathbf{IT}(\mathbb{N})$ by a data-predicative derivation with data-positive induction iff P is computable in linear space. And for any word algebra $\mathbb{A}(C)$, such as $\{0,1\}^*$, P is provable in $\mathbf{IT}(C)$ by a data-predicative derivation with data-positive induction iff P is computable in polynomial time on a Turing machine.

Note the generic character of Theorem 5. Here we get different complexity classes (according to common separation conjectures) depending on the underlying data. This is because the coding of one algebra in another exceeds here the computational complexity under consideration, contrary to broader classes, such as the Kalmar-elementary functions.

2.3 Classical vs. constructive logic

The relations between the constructive and classical versions of the results above are slightly more complex than in unrestricted intrinsic theories. In [20, §3.6] we exhibited a trivial proof that a function provable in an intrinsic theory based on classical logic, is already provable in that theory based on minimal logic. However, that proof uses a formula-substitution that may convert data-predicative formulas to ones that are no longer data-predicative, and similarly for data-non-positive formulas. More subtle proofs are therefore needed here. Below we present a rather direct proof of Proposition 2 for the constructive logic case. However, as of this writing we do not know whether this proof can be modified to apply also to classical logic; we therefore give a separate proof for the classical logic case, in Appendix I, using a different technique, and somewhat more complex.

The difference between the classical and constructive variants of the theories is manifest with non-data-positive induction. On the one hand we have:

PROPOSITION **6** *Based on classical logic, data-non-positive induction is equivalent to data-non-negative induction.*

The proof here is, for $\mathbb{N}$, similar to the proof that $\mathbf{\Pi}^0_1$-induction is equivalent to $\mathbf{\Sigma}^0_1$-induction (see e.g. [6] or [27]). A bit more work is needed for other data systems. We don't know if Proposition 6 remains true for data-predicative derivations.

In contrast to Proposition 6 we have:

PROPOSITION **7** *Every function provable in $\mathbf{IT}(C)$, based on constructive logic, and with induction for data-non-positive formulas, is explicitly definable from the constructors of C.*

This is a consequence of Theorem 18 below.

2.4 Relativized results

All results above can be relativized, as follows. If f is an r-ary function-identifier, we write

$$\mathrm{Tot}[\mathsf{f}] \quad \equiv_{\mathrm{df}} \quad \forall x_1 \ldots x_r.\, \mathbf{D}(x_1) \wedge \cdots \mathbf{D}(x_r) \rightarrow \mathbf{D}(\mathsf{f}(\vec{x}))$$

Suppose $\mathbb{A}(C)$ is an inductive data-system, which without loss of generality and to avoid cluttered notations we assume single-sorted. Let $g_1, g_2 \ldots\}$ be discourse-level parameters ranging over functions over $\mathbb{A}(C)$, and $\mathbf{g}_1, \mathbf{g}_2$ be corresponding formal function-identifiers (thus $\underline{arity}(\mathbf{g}_i) = \underline{arity}(g_i)$). The aforementioned results then hold with "primitive recursive" replaced by "primitive recursive in $g_1, \ldots$", and with "provable" replaced by "provable from $\mathrm{Tot}[\mathbf{g}_1]$, $\mathrm{Tot}[\mathbf{g}_2], \ldots$."

Relativization of the results is of interest when embedding traditional first order theories in intrinsic theories. For example, embedding Peano's Arithmetic in $\mathbf{IT}(\mathbb{N})$, as in [20], introduces into $\mathbf{IT}(\mathbb{N})$ addition and multiplication as new primitives. This augmentation is inconsequential in virtually any application, since addition and multiplication are provable functions in very weak variants $\mathbf{IT}(\mathbb{N})$. However, not so in $\mathbf{IT}(\mathbb{N})$ with induction restricted to data-non-positive formulas. The relativized analog of Proposition 7 is then worth independent consideration:

PROPOSITION **8** *The functions provable in* $\mathbf{IT}(C)$*, based on constructive logic, and with induction for data-non-positive formulas, from the statements of totality of functions $f_1 \ldots f_k$, are precisely the functions explicitly definable from the constructors of C and $f_1 \ldots f_k$.*

In particular, the provably recursive functions of Heyting's Arithmetic with induction restricted to data-non-negative formulas, are precisely the functions explicitly definable from $0, 1, +$ *and* $\times$.

The latter part of Proposition 8 improves one of the results of [29].[3]

Another illustration of the crucial difference between constructive and classical version of the theories, when induction formulas are structurally restricted, is this:

THEOREM **9** *The provable functions of constructive* $\mathbf{IT}(\mathbb{N})$ *with induction restricted to prenex formulas are (classically, whence constructively) provable using* $\mathbf{\Pi}_2$*-induction.*

It is easy to extract from Theorem 18 below a proof outline: prenex formulas are mapped to types of order 2, and functions definable by order-2 primitive recursion are well known to be the same as the functions provable by $\mathbf{\Pi}_2$-induction.[4]

3 FROM COMPUTATIONAL COMPLEXITY TO PROVABILITY

We start with the forward implication of Proposition 1.

[3]The latter states that the provably recursive functions of $\mathbf{HA}$ based on $\mathbf{\Pi}^0_1$-induction are bounded by polynomials, and are the same as the provably recursive functions of $\mathbf{HA}$ with induction for formulas in either one of the classes $\neg\mathbf{\Sigma}_1$, $\neg\neg\mathbf{\Sigma}_1$, $\neg\mathbf{\Pi}_1$, or $\neg\neg\mathbf{\Pi}_1$. It should be noted, though, that induction for formulas with no strictly-positive data information is useless only when it comes to proving program termination. Indeed, one can prove by induction on $\mathbf{\Pi}_1$-formulas, i.e. of the form $\forall x\ \mathbf{N}(x) \rightarrow \boldsymbol{\alpha}$, $\boldsymbol{\alpha}$ quantifier-free, that addition is commutative, a result which cannot be proved using only $\mathbf{\Sigma}_1$-induction, even though the latter theory is so much more powerful than the former with respect to proving program termination.

[4]This follows, e.g. from Gödel's "Dialectica" interpretation [5]. A consequence of Theorem 9 is that the functions provably-recursive in $\mathbf{HA}$ with prenex induction are provably-recursive in $\mathbf{HA}$ with $\mathbf{\Pi}_2$ induction, which is the main theorem of [29].

LEMMA **10** *Every function over $\mathbb{A}(C)$ definable by simultaneous recurrence is provable in $\mathbf{IT}(C)^+$; moreover, the derivation uses induction only for conjunctions of atomic formulas.*

In particular, all primitive recursive functions over $\mathbb{N}$ are provable in $\mathbf{IT}(\mathbb{N})$ using such restricted induction.

Proof. See [20, Proposition 6]. ⊣

LEMMA **11** *Addition and multiplication are provable by data-predicative derivations, with induction over atomic formulas. Exponentiation is provable by a data-predicative derivation.*

Proof. We use the following programs for addition, multiplication, and base-2 exponentiation:

$$\begin{array}{rcl} x+\mathbf{0} &=& x\\ x+\mathsf{s}y &=& \mathsf{s}(x+y)\end{array}\qquad \begin{array}{rcl} x\times\mathbf{0} &=& \mathbf{0}\\ x\times\mathsf{s}y &=& (x\times y)+x\end{array}\qquad \begin{array}{rcl} \mathsf{e}(\mathbf{0},y) &=& \mathsf{s}y\\ \mathsf{e}(\mathsf{s}x,y) &=& \mathsf{e}(x,\mathsf{e}(x,y))\end{array}\qquad \mathsf{exp}(x)=\mathsf{e}(x,\mathbf{0}).$$

The corresponding proofs are as follows, where for readability we omit uses of the programs' equations, and use instead double-bars to indicate such uses (via equational rules).

$$\dfrac{\mathsf{N}(y)\qquad \genfrac{}{}{1.5pt}{0}{\mathsf{N}(x)}{\mathsf{N}(x+\mathbf{0})}\qquad \genfrac{}{}{1.5pt}{0}{\dfrac{\mathsf{N}(x+z)}{\mathsf{N}(\mathsf{s}(x+z))}}{\mathsf{N}(x+\mathsf{s}z)}}{\mathsf{N}(x+y)}$$

$$\dfrac{\mathsf{N}(y)\qquad \genfrac{}{}{1.5pt}{0}{\mathsf{N}(\mathbf{0})}{\mathsf{N}(x\times\mathbf{0})}\qquad \genfrac{}{}{1.5pt}{0}{\begin{array}{c}\mathsf{N}(x)\qquad\quad \mathsf{N}(x+z)\\ \cdots\\ \mathsf{N}((x\times z)+x)\end{array}}{\mathsf{N}(x\times(\mathsf{s}z))}}{\mathsf{N}(x\times y)}$$

$$\genfrac{}{}{1.5pt}{0}{\genfrac{}{}{1.5pt}{0}{\dfrac{\mathsf{N}(x)\qquad \genfrac{}{}{1.5pt}{0}{\genfrac{}{}{1.5pt}{0}{\dfrac{\mathsf{N}(y)}{\mathsf{N}(\mathsf{s}y)}}{\mathsf{N}(\mathsf{e}(\mathbf{0},y))}}{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(\mathbf{0},y)))}\qquad \genfrac{}{}{1.5pt}{0}{\genfrac{}{}{1.5pt}{0}{\dfrac{\dfrac{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(u,y))}{\mathsf{N}(\mathsf{e}(u,y))\to\mathsf{N}(\mathsf{e}(u,\mathsf{e}(u,y)))}\qquad \dfrac{\dfrac{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(u,y))}{\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(u,y))}\quad \mathsf{N}(y)}{\mathsf{N}(\mathsf{e}(u,y))}}{\mathsf{N}(\mathsf{e}(u,\mathsf{e}(u,y)))}}{\mathsf{N}(\mathsf{e}(\mathsf{s}u,y))}}{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(\mathsf{s}u,y)))}}{\forall y\,(\mathsf{N}(y)\to\mathsf{N}(\mathsf{e}(x,y))}}{\mathsf{N}(\mathsf{e}(x,\mathbf{0}))}}{\mathsf{N}(\mathsf{exp}(x))}$$

Note that here the induction formula has N in both positive and negative positions. ⊣

Building on the construction of Proposition 11, we can prove the forward direction of Theorems 4 and 5.

PROPOSITION **12** *Let $\mathbb{A}(C)$ be a data-system, f a function computable by a register machine over $\mathbb{A}(C)$.*

1. *If the register machine for f terminates in time that is Kalmar-elementary in the height of the input, then f is provable in* $\mathbf{IT}(C)$ *by a data-predicative derivation, in minimal logic.*
2. *If the register machine for f terminates in time polynomial in the height of the input, then f is provable in* $\mathbf{IT}(C)$ *by a data-predicative derivation, using induction on the conjunction of atomic formulas.*

Proof Outline. As in [17, §3.3] we refer to the coding of register-machine configurations by tuple of terms of $\mathbb{A}(C)$. Given a deterministic m-register machine M over $\mathbb{A}(C)$, there are functions $\tau_0, \tau_1 \dots, \tau_m : \mathbb{A}^{m+1} \rightarrow \mathbb{A}$ such that, if M has a transition rule for state s, then $(s, [u_1, \dots, u_m]) \vdash_M (s', [u'_1, \dots, u'_m])$ iff $\tau_i(\#s, u_1, \dots, u_m) = u'_i$ for $i = 1 \dots m$, and $\tau_0(\#s, u_1, \dots, u_m) = \#s'$; and if M has no such transition, then $\tau_i(u_0 \dots u_m) = u_i$. The transition functions τ_i are provable, in the strong sense that $\bigwedge_i \mathbf{D}(u_i) \rightarrow \mathbf{D}(\tau_i(\vec{u})$ has a data-predicative derivation, that uses only degenerate instances of data-elimination.

As in the proof of [17, Lemma 3.4], we replace the successor function in the derivations for addition, multiplication and exponentiation above, by the tuples $\langle \tau_0 \dots \tau_m \rangle$, and conclude the every computation that terminates with the given bounds are provable with the corresponding predicativity restrictions. $\dashv$

4 FROM PROVABILITY TO COMPUTATIONAL COMPLEXITY

By the well known Curry-Howard morphisms, natural deductions can be viewed as λ-terms [24, 11]. In [13, 14] we presented a version of such mapping which yields directly an applicative program for a function f from a derivation for the provability of f in various formalisms. We summarize here a similar construction for intrinsic theories.

4.1 Typed lambda calculus with recurrence

We shall map derivations of $\mathbf{IT}(C)$ to terms of an applied simply typed λ-calculus. We focus attention first on single-sorted data systems C, and write $\mathbf{D}$ for the unique sort.

Let $\mathbf{1\lambda}$ be the simply typed lambda calculus with type products, and with pairing and projections as term-constructs. We let $\rightarrow$ associate to the right, and write $\rho_1, \dots, \rho_r \rightarrow \tau$ for $\rho_1 \rightarrow \cdots \rightarrow \rho_r \rightarrow \tau$ as well as for $\rho_1 \times \cdots \times \rho_r \rightarrow \tau$. If all ρ_i's are the same type ρ, we write $\rho^r \rightarrow \tau$ for the above. We write $\langle t, t' \rangle$ for pairing of the terms t, t', and π_0, π_1 for the two projection functions. The computational rules are the usual *β-reduction,* contracting $(\lambda x.t)s$ to $\{s/x\}t$, and *pair-reduction,* contracting $\pi_i\langle t_0, t_1 \rangle$ to t_i $(i = 0, 1)$.

Let $C = \{\mathbf{c}_1 \dots \mathbf{c}_k\}$ be a single-sorted data-system, with $\mathbf{c}_i$ of arity r_i. The extension $\mathbf{1\lambda}(C)$ of $\mathbf{1\lambda}$ is defined as follows. Each $\mathbf{c}_i$ is admitted as a constant, of type $\iota^{r_i} \rightarrow \iota$. For each type τ, we also have a constant $\mathbf{R}^C_\tau = \mathbf{R}_\tau$ of type $\sigma_1[\tau], \dots, \sigma_k[\tau], \iota \rightarrow \tau$, where $\sigma_i[\tau] =_{\text{df}} \tau^{r_i} \rightarrow \tau$. Aside from the β-reduction and pair-reduction, $\mathbf{1\lambda}(C)$ has for each type τ a rule of *recurrence in type* τ:

$$\mathbf{R}_\tau M_1 \cdots M_k(\mathbf{c}_i A_1 \dots A_{r_i}) \Rightarrow M_i A'_1 \cdots A'_{r_i} \qquad \text{where } A'_j =_{\text{df}} \mathbf{R}_\tau M_1 \cdots M_k A_j$$

A term is *normal* if no subterm can be reduced.

The Tait-Prawitz method is easily applicable to $\mathbf{1\lambda}(C)$, yielding

LEMMA **13** *Every reduction sequence in* $\mathbf{1\lambda}(C)$ *terminates.* $\dashv$

Note that each element of $\mathbb{A}(C)$ is represented in $\mathbf{1\lambda}(C)$ by itself (modulo currying), and that every closed normal term of type ι is an element of $\mathbb{A}(C)$. Thus, every expression of type $\iota^r \to \iota$ represents an r-ary function over $\mathbb{A}(C)$. Clearly, the constant $\mathbf{R}_\tau$ denotes the operation of *recurrence*, i.e. *iteration with parameters,* over type τ, that is, the function $\langle g_1 \ldots g_k \rangle \mapsto f$, where f is defined by $f(\mathbf{c}_i(a_1 \ldots a_{r_i})) = g_i(f(a_1) \ldots f(a_{r_i}))$ $(i = 1 \ldots k)$. It follows that the functions over $\mathbb{A}(C)$ represented in $\mathbf{1\lambda}(C)$ are precisely the functions generated by explicit definitions and recurrence in finite types.

Various extensions of this calculus are discussed in [20].

4.2 Natural deduction derivations as applicative programs

Let C be a single-sorted data-system. We map ND derivations of $\mathbf{IT}(C)$, based on minimal logic, to terms of $\mathbf{1\lambda}(C)$, as follows. Our mapping is in the spirit of Curry-Howard's formula-as-type analogy, but with a twist. In [14] we defined a mapping that disregards the first order part of formulas, notably application of predicates to terms, and first order universal quantification. Our present mapping also disregards equality, and is therefore potentially oblivious to large parts of derivations. In fact, no proper term will correspond to a derivation of an equation. To convey this approach, we use an auxiliary atomic type $\perp$ (for "undefined"), as well as an auxiliary atomic term $\emptyset$, of type $\perp$.

We first define recursively a mapping κ from formulas to types:

- $$\kappa(\mathbf{D}(\vec{t})) =_{\text{df}} \mathbf{D}$$
- $$\kappa(t = t') =_{\text{df}} \perp$$
- $$\kappa(\psi \to \chi) =_{\text{df}} \begin{cases} \kappa(\chi) & \text{if } \kappa(\psi) = \perp \\ \perp & \text{if } \kappa(\chi) = \perp \\ \kappa\psi \to \kappa\chi & \text{otherwise} \end{cases}$$
- $$\kappa(\psi \wedge \chi) =_{\text{df}} \begin{cases} \kappa(\chi) & \text{if } \kappa(\psi) = \perp \\ \kappa(\psi) & \text{if } \kappa(\chi) = \perp \\ \kappa\psi \times \kappa\chi & \text{otherwise} \end{cases}$$
- $$\kappa(\forall x.\psi) =_{\text{df}} \kappa\psi$$

We call a formula φ *data-negative* if $\kappa(\varphi) = \perp$, *data-positive* otherwise. Thus, φ is data-positive iff it has a strictly-positive atomic subformula $\mathbf{D}(\vec{t})$. We call a derivation $\mathcal{D}$ data-positive (data-negative) if its derived formula is data-positive (data-negative, respectively).

We proceed to define recursively a mapping from ND derivations of $\mathbf{IT}(C)$, based on minimal logic and using only the closed-version of C-Induction, to terms of $\mathbf{1\lambda}(C)$. Without danger of ambiguity we write κ also for this mapping. If $\mathcal{D}$ is a derivation from labeled open assumptions $\psi_1^{j_1}, \ldots, \psi_q^{j_q}$ to conclusion φ, then $\kappa\mathcal{D}$ will be a term of type $\kappa\varphi$, with free variables among $x_{j_1} \ldots x_{j_q}$, of types $\kappa\psi_1 \ldots \kappa\psi_q$, respectively.[5] The type $\perp$ and the object $\emptyset$ will be used in the definition only when equality is present.

[5]We posit a concrete syntax for natural deductions, which assigns a common numeric label to each assumption class, i.e. the open assumptions that are closed jointly by an inference. Distinct labels are assigned to different assumption classes.

The term $\kappa\mathcal{D}$ is defined by recurrence on $\mathcal{D}$, as follows, using momentarily the convention that if $\mathcal{D}$ is a derivation, then $\mathcal{D}_0, \mathcal{D}_1, \ldots$ are $\mathcal{D}$'s immediate sub-derivations, in that order.

1. If $\mathcal{D}$ is data-negative, then $\kappa\mathcal{D} = \emptyset$.

The following cases refer to a data-positive $\mathcal{D}$.

2. If $\mathcal{D}$ is a labeled open assumption ψ^j, then $\kappa\mathcal{D} = x_j^{\kappa\psi}$ (the j-th variable of type $\kappa\psi$).
3. If $\mathcal{D}$ derives $\varphi_0 \wedge \varphi_1$ by conjunction introduction, then $\kappa\mathcal{D} = \langle \kappa\mathcal{D}_0, \kappa\mathcal{D}_1 \rangle$ if both $\mathcal{D}_0$ and $\mathcal{D}_1$ are data-positive. If only $\mathcal{D}_i$ is data positive, then $\kappa\mathcal{D} = \kappa\mathcal{D}_i$.
4. If $\mathcal{D}$ derives φ_i from $\varphi_0 \wedge \varphi_1$ ($i = 0$ or 1), then $\kappa\mathcal{D} = \pi_i \kappa\mathcal{D}_0$ if both φ_0 and φ_1 are data-positive, $\kappa\mathcal{D} = \kappa\mathcal{D}_0$ otherwise.
5. Suppose $\mathcal{D}$ derives $\psi \rightarrow \varphi$ by implication introduction, closing labeled assumption ψ^j. If ψ is data-positive, then $\kappa\mathcal{D} = \lambda x_j^{\kappa\psi}.\, \kappa\mathcal{D}_0$; otherwise, $\kappa\mathcal{D} = \kappa\mathcal{D}_0$.
6. Suppose $\mathcal{D}$ derives φ by implication elimination, from $\psi \rightarrow \varphi$ and ψ. Since $\mathcal{D}$ is data-positive, the sub-derivation $\mathcal{D}_0$ of $\psi \rightarrow \varphi$ is also data-positive. If ψ is data-positive as well, then $\kappa\mathcal{D} = (\kappa\mathcal{D}_0)(\kappa\mathcal{D}_1)$; otherwise, $\kappa\mathcal{D} = \kappa\mathcal{D}_0$
7. If the main inference of $\mathcal{D}$ is $\forall$-introduction, $\forall$-elimination, or Replacement, then $\kappa\mathcal{D} = \kappa\mathcal{D}_0$.
8. If $\mathcal{D}$ derives $\mathbf{D}(\mathbf{c}_i(t_1 \ldots t_{r_i}))$ by the generative rule for $\mathbf{c}_i$, from sub-derivations $\mathcal{T}_1 \ldots \mathcal{T}_{r_i}$, then $\kappa\mathcal{D} = \mathbf{c}_i(\kappa\mathcal{T}_1, \cdots, \kappa\mathcal{T}_{r_i})$.[6]
9. If $\mathcal{D}$ derives $\varphi[t]$ by the closed-version of C-induction from $\mathbf{D}(t)$ and $Cl_{\mathbf{c}_j}[\lambda x \varphi]$ ($j = 1 \ldots k$), then $\kappa\mathcal{D} = \mathbf{R}_{\kappa\varphi}(\kappa\mathcal{D}_1) \cdots (\kappa\mathcal{D}_k)(\kappa\mathcal{D}_0)$. Note that φ here is assumed data-positive, hence all sub-derivations are data positive.[7]

LEMMA **14** *If $\mathcal{D}$ is a derivation of φ from labeled open assumptions $\psi_1^{j_1}, \ldots, \psi_m^{j_m}$, then $\kappa\mathcal{D}$ is a term of type $\kappa\varphi$, with free variables among $x_{j_1}, \ldots, x_{j_m}$ of types $\kappa\psi_1, \ldots \kappa\psi_m$, respectively.*

Proof. By a straightforward structural induction on $\mathcal{D}$. $\dashv$

4.3 Function provability and recurrence in higher type

LEMMA **15** *For every $a \in \mathbb{A}(C)$ there is a normal deduction $\mathcal{T}_a$ in $\mathbf{IT}(C)$ deriving $\mathbf{D}(a)$, such that $\kappa\mathcal{T}_a = a$.*

Proof. Trivial induction on a. $\dashv$

LEMMA **16** *If $\mathcal{D}$ reduces to $\mathcal{D}'$ in $\mathbf{IT}(C)$, then either $\kappa\mathcal{D} = \kappa\mathcal{D}'$, or $\kappa\mathcal{D}$ reduces to $\kappa\mathcal{D}'$ in $\mathbf{1}\boldsymbol{\lambda}(C)$.*

[6]Note that in this case all sub-derivations $\mathcal{T}_i$ are data-positive.

[7]This is no longer true for multi-sorted C, where C-Induction may be used to derive a data-positive $\varphi_i[t]$ with some other inductive formulas $\varphi_j[x]$ being data-negative.

Proof. Straightforward inspection of the reductions. Note that for the case of Replacement reductions for Induction it is important that we allow non-atomic eigen-formulas in instances of Replacement.[8]

LEMMA **17** *Let P be an equational program. If $\mathcal{D}$ is a normal derivation of* $\mathbf{IT}(C)$*, deriving an atomic formula $\mathbf{D}_i(t)$ from $\bar{P}$, where t is closed, then $\kappa\mathcal{D}$ is a base term and $t = \kappa\mathcal{D}$ is derived from P in equational logic.*

Proof. By basic properties of normal derivations $\mathcal{D}$ is without C-induction, and so every formula in $\mathcal{D}$ is atomic, or of the form $\forall\vec{x}.\, t = t'$ for some equation $t = t'$ in P.

We proceed to prove the Lemma by structural induction on $\mathcal{D}$. If $\mathcal{D}$ is a singleton derivation, then it must be $\mathbf{D}_i(\mathbf{c})$, where $\mathbf{c}$ is a 0-ary constructor of C. The Lemma holds trivially.

If $\mathcal{D}$ is

$$\frac{\begin{matrix}\mathcal{D}_0 & \mathcal{D}_1 \\ \mathbf{D}_i(t') & t' = t\end{matrix}}{\mathbf{D}_i(t)}$$

then $\kappa\mathcal{D} = \kappa\mathcal{D}_0$, where by IH $\kappa\mathcal{D}_0$ is a base term, with $P \vdash^{=} t' = \kappa\mathcal{D}_0$. Also, $P \vdash^{=} t' = t$. Thus, $\kappa\mathcal{D}$ is a base term, and $P \vdash^{=} t = \kappa\mathcal{D}$.

Finally, if $\mathcal{D}$ is

$$\frac{\begin{matrix}\mathcal{T}_1 & & \mathcal{T}_{r_i} \\ \mathbf{D}_{p_{i1}}(t_1) & \cdots & \mathbf{D}_{p_{ir_i}}(t_{r_i})\end{matrix}}{\mathbf{D}_{q_i}(\mathbf{c}_i(t_1 \ldots t_{r_i}))}$$

then $\kappa\mathcal{D} = \mathbf{c}_i(\kappa\mathcal{T}_1, \ldots, \kappa\mathcal{T}_{r_i})$. By IH $\kappa\mathcal{T}_j$ is a base term, where $P \vdash^{=} t_j = \kappa\mathcal{T}_j$ $(j = 1 \ldots r_i)$. Therefore $\kappa\mathcal{D} = \mathbf{c}_i(\kappa\mathcal{T}_1, \ldots, \kappa\mathcal{T}_{r_i})$ is a base term, and $P \vdash^{=} \mathbf{c}_i(t_1 \ldots t_{r_i}) = \kappa\mathcal{D}$. ⊣

THEOREM **18** [Representation] *Let $(P, \mathbf{f})$ be a program computing a function f over $\mathbb{A}(C)$. If $\mathcal{D}$ is a deduction of* $\mathbf{IT}(C)$ *deriving $\mathbf{D}(\vec{x}) \rightarrow \mathbf{D}(\mathbf{f}(\vec{x}))$ from $\bar{P}$, then $\kappa\mathcal{D}$ represents f in $\mathbf{1\lambda}(C)$.*

Proof. Without loss of generality, let f be unary. Given $a \in \mathbb{A}(C)$ let $\mathcal{D}_a$ be the result of substituting a for free occurrences of x in $\mathcal{D}$. We have $\kappa\mathcal{D}_a = \kappa\mathcal{D}$ trivially from the definition of κ. Let $\mathcal{T}_a$ be the straightforward derivation of $\mathbf{D}(a)$, using the data introduction rules for C. Then, for

$$\mathcal{D}_{fa} \;=_{\rm df}\; \frac{\begin{matrix}\mathcal{D}_a & \mathcal{T}_a \\ \mathbf{D}(a) \rightarrow \mathbf{D}(\mathbf{f}(a)) & \mathbf{D}(a)\end{matrix}}{\mathbf{D}(\mathbf{f}a)}$$

we have $\kappa(\mathcal{D}_{fa}) = (\kappa\mathcal{D}_a)(\kappa\mathcal{T}_a)$. By Lemma 13 $\mathcal{D}_{fa}$ reduces to a normal derivation $\mathcal{D}'_{fa}$, and by Lemma 17 $\kappa(\mathcal{D}'_{fa})$ is a base term. We thus have

$$\begin{array}{rcll} (\kappa\mathcal{D})(a) & = & (\kappa\mathcal{D}_a)(\kappa\mathcal{T}_a) & \text{by Lemma 15} \\ & = & \kappa\mathcal{D}_{fa} & \\ & = & \kappa\mathcal{D}'_{fa} & \text{by Lemma 16} \\ & = & f(a) & \end{array}$$

[8]Without this stipulation $\kappa\mathcal{D}$ would not reduce to $\kappa\mathcal{D}'$, but to a term to which $\kappa\mathcal{D}'$ η-reduces. The gist of our results would be preserved, but in a less transparent setting. Moreover, we must refer to Replacement reduction over Induction (data elimination), so using Replacement reduction over logical elimination rules only enhances uniformity and symmetry.

Thus $\kappa\mathcal{D}$ represents f. $\dashv$

4.4 Data-positive induction and primitive recursion

An immediate consequence of Theorem 18 is the backward direction of Theorem 3:

COROLLARY **19** *Every function provable in* $\mathbf{IT}^{+}(\mathbb{N})$ *(or* $\overline{\mathbf{IT}}^{+}(\mathbb{N})$*) is primitive recursive.*

Proof. By Theorem 18 every function provable in $\mathbf{IT}^{+}(\mathbb{N})$ is representable in $\mathbf{1}\boldsymbol{\lambda}(\mathbb{N})$ by a term with the recurrence operator used for types of the form $\mathbf{N}^k$ ($k \geqslant 1$), i.e. the functions defined by simultaneous primitive recursion at based type. All such functions are primitive recursive (see e.g. [23].) $\dashv$

4.5 Data-predicative proofs

Proof outline. In [20] we exhibited proofs for addition, multiplication, and exponentiation. The derivations there for addition and multiplication are data-predicative and with data-positive induction. The derivation for exponentiation is data-predicative. It is easy to see that the provable functions (under any one of the paradigms above) are closed under bounded recurrence and composition. It follows that every function defined from addition and multiplication by bounded primitive recursion, i.e. function in Grzegorczyk's class $\mathcal{E}_2$ = linear-space for numeric functions, is provable by a data-predicative derivation of $\mathbf{IT}(\mathbb{N})$ with data-positive induction. This establishes (3). (1) and (2) are analogous.

More complex and interesting are the converse implications. We use here the results of [19]. The mapping κ of proofs to functions, developed for Theorem 18 above, maps data-predicative derivations to $\mathbf{1}\boldsymbol{\lambda}(C)$-terms dubbed *input-driven* in [19], and shown there to define only functions computable in elementary resources, completing the proof of (1). Similarly, κ maps data-predicative derivations with data-positive induction to input-driven $\mathbf{1}\boldsymbol{\lambda}(C)$-terms with recurrence for first-order types only, which by the results of [19] yield (2) and (3). **End of Proof Outline.**

5 APPENDIX: THEOREM 3 FOR CLASSICAL LOGIC

Theorem 3 above refers to theories based on constructive logic, since we show that every provable function is primitive recursive by a Curry-Howard mapping that we are unable as of this writing to adapt to classical logic. However, the theorem holds for classical logic as well, as we show in this Appendix using a different method, namely a direct proof-theoretic analysis of $\overline{\mathbf{IT}}^{+}(\mathbb{N})$. The analysis uses a novel technique that recursively unfolds inductions in data-closed derivations, without addressing the normalization of $\overline{\mathbf{IT}}^{+}(\mathbb{N})$ in general.

We shall freely use common concepts and terminology for natural deduction calculi, as presented e.g. in [28]. We call a derivation of $\overline{\mathbf{IT}}(\mathbb{N})$ *simply-normal* if it is induction-free and normal (in the sense of first order logic). If $\mathcal{D}$ is a natural deduction derivation, we write $|\mathcal{D}|$ for its height.

5.1 Converting data-non-negative proofs to induction-free proofs

Suppose $\mathcal{D}$ is a (classical logic, Gentzen-style) natural deduction of $\overline{\mathbf{IT}}(\mathbb{N})$, deriving a formula A from (open) assumptions $B_1 \ldots B_m$. By the normalization theorem for Gentzen's natural deduction (see e.g. [28]), we may assume that $\mathcal{D}$ is normal. If $\mathcal{D}$ is in intuitionistic logic, then each formula-occurrence in it must be a subformula of either A, some B_i, or some eigen-formula of an instance of the induction rule. If classical logic is used, then we might also have in $\mathcal{D}$ negations of formulas as above; each such negation must be an assumption of $\mathcal{D}$ closed by the $\perp$-rule, and the major premise of Implication-Elimination.

We dub an (open) assumption of a derivation $\mathcal{D}$ *contradiction-assumption* if it is negated and the major premise of Implication-Elimination.

Now suppose that $\mathcal{D}$ is a derivation of $\mathbf{IT}^{+}(\mathbb{N})$ that derives a data-non-negative formula A from data-non-negative formulas and contradiction-assumptions. As noted above, we may assume without loss of generality that A and the data-non-negative assumptions, as well as the eigen-formulas of instances of induction in $\mathcal{D}$, are all in prenex-disjunctive form. Since subformulas of data-non-negative prenex-disjunctive formulas are also data-non-negative, all formulas in $\mathcal{D}$, with the possible exception of contradiction-assumptions, are data-non-negative. We call such derivations *data-non-negative*. Finally, a data-non-negative derivation is *data-closed* if all assumptions are $\mathbf{N}$-free.

The usual normalization procedure for first order natural deductions [22, 28] yields:

LEMMA **20** *There is a primitive recursive mapping that, modulo canonical encoding of syntax, yields from induction-free derivations of* $\overline{\mathbf{IT}}(\mathbb{N})$ *equivalent simply-normal derivations.*[9] *In particular, there is a primitive recursive function δ such that for every induction-free derivation $\mathcal{D}$ there is an equivalent simply-normal $\mathcal{D}'$, with $|\mathcal{D}'| \leqslant \delta(\mathcal{D})$.*

LEMMA **21** *Suppose $\mathcal{D}$ is a simply-normal data-closed derivation of a data-positive formula A. Then the main inference of $\mathcal{D}$ is not an elimination.*

In particular, if a simply-normal data-closed $\mathcal{D}$ derives an atomic formula $\mathbf{N}(t)$, then the main inference of $\mathcal{D}$ must be Substitution or Data-Introduction.

[9]The procedure is in fact at the fourth level of Grzegorczyk's hierarchy.

Proof. By induction on $\mathcal{D}$. If the main inference of $\mathcal{D}$ is an elimination, then that inference's major premise, call it F, is also data-positive, and derived by a simply-normal data-closed derivation $\mathcal{D}_0$. (Note that F cannot be a contradiction-assumption, because A is data-positive, whence cannot be $\bot$.) Moreover, since $\mathcal{D}$ is normal, the main inference of $\mathcal{D}_0$ cannot be an introduction; and since F is not atomic, that inference can only be an elimination, contradicting induction assumption. $\dashv$

LEMMA **22** *Suppose that*

$$\mathcal{D} \quad = \quad \dfrac{\begin{matrix} \mathcal{D}_N \\ \mathbf{N}(t) \end{matrix} \quad \begin{matrix} \mathcal{D}_0 \\ A[\mathbf{0}] \end{matrix} \quad \begin{matrix} \{A[x]\} \\ \mathcal{D}_s[x] \\ A[\mathsf{s}x] \end{matrix}}{A[t]}$$

where $\mathcal{D}_N$ is simply-normal and data-closed, $\mathcal{D}_0$ is simply-normal, and where for every simply-normal and data-closed derivation $\begin{matrix}\mathcal{E}\\ A[\bar{n}]\end{matrix}$ there is a simply-normal and data-closed derivation equivalent to

$$\begin{matrix} \mathcal{E} \\ A[\bar{n}] \\ \mathcal{D}_s[\bar{n}] \\ A[\mathsf{s}\bar{n}] \end{matrix}$$

of height $\leqslant \zeta(|\mathcal{E}|)$, where ζ is primitive recursive and increasing. Let ℓ be the number of instances of Successor-Introduction in $\mathcal{D}_N$, m the height of $\mathcal{D}_N$, and a the number of logical symbols in A. Then there is a simply-normal and data-closed derivation $\mathcal{D}^o$ equivalent to $\mathcal{D}$, of height $\leqslant \zeta_m^{[\ell]}(|\mathcal{D}_0|)$, where $\zeta_m(x) =_{\mathrm{df}} \delta(\zeta(x + 2ma))$.[10]

Proof. By induction on $|\mathcal{D}_N|$. By Lemma 21, $\mathbf{N}(t)$ can be derived only by Data-Introduction or Substitution. If the inference is a Zero-Introduction, i.e. $\mathcal{D}_N$ is the singleton $\mathbf{N}(\mathbf{0})$, take $\mathcal{D}^o = \mathcal{D}_0$. Here $n = m = 0$, and indeed $|\mathcal{D}_0| = \zeta_0^{[0]}(|\mathcal{D}|)$.

Suppose that the main inference of $\mathcal{D}_N$ is Successor-Introduction,

$$\mathcal{D}_N \quad = \quad \dfrac{\begin{matrix} \mathcal{D}_{N0} \\ \mathbf{N}(t') \end{matrix}}{\mathbf{N}(\mathsf{s}t')}$$

where t is $\mathsf{s}t'$. $\mathcal{D}$ is then equivalent to

$$\mathcal{D}_1 \quad = \quad \begin{matrix} \mathcal{D}' \\ A[t'] \\ \mathcal{D}_s[t'] \\ A[\mathsf{s}t'] \end{matrix}$$

where

$$\mathcal{D}' \quad = \quad \dfrac{\begin{matrix} \mathcal{D}_{N0} \\ \mathbf{N}(t') \end{matrix} \quad \begin{matrix} \mathcal{D}_0 \\ A[\mathbf{0}] \end{matrix} \quad \begin{matrix} \{A[x]\} \\ \mathcal{D}_s \\ A[\mathsf{s}x] \end{matrix}}{A[t']}$$

By induction assumption, $\mathcal{D}'$ can be mapped to an equivalent simply-normal derivation of height $\leqslant \zeta_m^{[\ell-1]}(|\mathcal{D}_0|)$, and by the Lemma's assumption about ζ it follows that $\mathcal{D}_1$ can be converted to an

[10]We write $f^{[q]}$ for the q'th iterate of the unary function f.

equivalent derivation of height $\leqslant \zeta(\zeta_m^{[n-1]}(|\mathcal{D}_0|))$, whence an equivalent simply-normal derivation of height

$$\begin{aligned} &\leqslant \delta\zeta\zeta_m^{\ell-1}(|\mathcal{D}_0|) \\ &\leqslant \delta\zeta\zeta_m^{\ell-1}(|\mathcal{D}_0|+2ma) \\ &= \zeta_m^{\ell}(|\mathcal{D}_0|) \end{aligned}$$

Finally, if the main inference of $\mathcal{D}_N$ is Substitution,

$$\mathcal{D}_N \quad = \quad \frac{\begin{array}{cc}\mathcal{D}_{N0} & \mathcal{D}_= \\ \mathbf{N}(t') & t{=}t'\end{array}}{\mathbf{N}(t)}$$

then $\mathcal{D}$ is equivalent to

$$\mathcal{D}_2 \quad = \quad \frac{\begin{array}{cc}\mathcal{D}' & \mathcal{D}_= \\ A[t'] & t{=}t'\end{array}}{\overline{A[t]}}$$

where $\mathcal{D}'$ is as above. By induction assumption $\mathcal{D}'$ can be mapped to an equivalent simply-normal derivation $\mathcal{D}'_1$, of height $\leqslant \zeta_{m-1}^{[n]}(|\mathcal{D}_0|)$. Consider

$$\mathcal{D}'' \quad = \quad \frac{\begin{array}{cc}\mathcal{D}'_1 & \mathcal{D}_= \\ A[t'] & t{=}t'\end{array}}{\overline{A[t]}}$$

The derived inference from $A[t']$ to $A[t]$, using $t{=}t'$ as an assumption, has height at most $2a$, as can be seen by a trivial induction on a. Thus

$$\begin{aligned} |\mathcal{D}''| &\leqslant \max(|\mathcal{D}_=|, |\mathcal{D}'_1|) + 2a \\ &\leqslant \max(|\mathcal{D}_=|, m) + 2a \\ &\leqslant \max(\zeta_{m-1}^{[n]}(|\mathcal{D}_0|), m) + 2a \\ &\leqslant \max(\zeta_{m-1}^{[n]}(|\mathcal{D}_0|+2a), m+2a) \\ &\leqslant \zeta_m^{[n]}(|\mathcal{D}_0|) \end{aligned}$$

$\dashv$

LEMMA **23** *Suppose that*

$$\begin{array}{ccc} B_1[\vec{x}] & \cdots & B_q[\vec{x}] \\ & \mathcal{D}[\vec{x}] & \\ & A[\vec{x}] & \end{array}$$

is a data-non-negative derivation of formula A, with data-positive assumptions among $B_1 \ldots B_q$ (and possibly equational assumptions and contradiction-assumptions), and the free variables in $B_1 \ldots B_q$ among $\vec{x} = x_1 \ldots x_p$. There is a primitive recursive function $\xi_{\mathcal{D}}$ that, modulo canonical coding of syntax, maps input numbers $\vec{n} = n_1 \ldots n_p$ and induction-free data-closed derivations[11]

$$\frac{\mathcal{C}_1}{B_1[\vec{n}]} \quad \cdots \quad \frac{\mathcal{C}_q}{B_q[\vec{n}]}$$

[11]Note that the derivations $\mathcal{C}_i$ are data-closed, and so the singleton derivation B_i is excluded.

to a derivation

$$\begin{array}{c}\mathcal{D}^*[\vec{n}]\\ A[\vec{n}]\end{array}$$

which is simply-normal and equivalent to

$$\mathcal{D}' \quad = \quad \begin{array}{ccc}\mathcal{C}_1 & & \mathcal{C}_q\\ B_1[\vec{n}] & \cdots & B_q[\vec{n}]\\ & \mathcal{D}[\vec{n}] & \\ & A[\vec{n}] & \end{array}$$

Proof. We proceed by induction on $\mathcal{D}$. If $\mathcal{D}$ is a single formula A, then we have two cases.

1. A is a data-positive formula, we have

$$\mathcal{D}' = \begin{array}{c}\mathcal{C}\\ A[\vec{n}]\end{array}$$

   where $\mathcal{C}$ is a data-closed induction-free derivation. The code of $\mathcal{D}'$ is trivially primitive recursive in $\vec{n}$ and $\mathcal{C}$, and applying the function δ of Lemma 20 yields $\mathcal{D}^*$.

2. A is not data-positive, in which case $\mathcal{D}^* = \mathcal{D}' = A[\vec{n}]$, and the Lemma's statement is trivial.

If the main inference of $\mathcal{D}$ is a data-introduction rule, an equational rule, or a logical rule that does not invlove closing an assumption, then the induction step is straightforward. We consider two typical cases, for Implication-Elimination and for $\forall$-Introduction. If the rule is Implication-Elimination,

$$\mathcal{D}[\vec{x}] \quad = \quad \dfrac{\begin{array}{c}B_1\cdots B_q\\ \mathcal{D}_0\\ A'\rightarrow A\end{array} \qquad \begin{array}{c}B_1\cdots B_q\\ \mathcal{D}_1\\ A'\end{array}}{A}$$

then, by induction assumption, we have a primitive recursive function that maps $\vec{n}$ and data-closed induction-free derivations

$$\dfrac{\mathcal{C}_1}{B_1[\vec{n}]} \quad \cdots \quad \dfrac{\mathcal{C}_q}{B_q[\vec{n}]}$$

to simply-normal data-closed derivations $\mathcal{D}_0^*$, equivalent to

$$\mathcal{D}_0' \quad = \quad \begin{array}{ccc}\mathcal{C}_1 & & \mathcal{C}_q\\ B_1[\vec{n}] & \cdots & B_q[\vec{n}]\\ & \mathcal{D}_0[\vec{n}] & \\ & A'[\vec{n}]\rightarrow A[\vec{n}] & \end{array}$$

and $\mathcal{D}_1^*$, equivalent to

$$\mathcal{D}_1' \quad = \quad \begin{array}{ccc}\mathcal{C}_1 & & \mathcal{C}_q\\ B_1[\vec{n}] & \cdots & B_q[\vec{n}]\\ & \mathcal{D}_1[\vec{n}] & \\ & A'[\vec{n}] & \end{array}$$

Then

$$\dfrac{\begin{array}{c}\mathcal{D}_0^*\\ A'[\vec{n}]\rightarrow A[\vec{n}]\end{array} \quad \begin{array}{c}\mathcal{D}_1^*\\ A'[\vec{n}]\end{array}}{A[\vec{n}]}$$

is induction-free and data-closed, and can be normalized by applying δ.

If

$$\mathcal{D}[\vec{x}] \qquad = \qquad \dfrac{\begin{matrix} B_1 \cdots B_q \\ \mathcal{D}_0 \\ A[v] \end{matrix}}{\forall u.\, A[u]}$$

then, by induction assumption, we have a primitive recursive function that maps $\vec{n}$ and data-closed induction-free derivations $\mathcal{C}_1 \ldots \mathcal{C}_q$ as above to a simply-normal data-closed derivation $\mathcal{D}_0^*$, equivalent to

$$\mathcal{D}'_0 \qquad = \qquad \begin{matrix} \mathcal{C}_1 & & \mathcal{C}_q \\ B_1[\vec{n}] & \cdots & B_q[\vec{n}] \\ & \mathcal{D}_0[\vec{n}] & \\ & A[v] & \end{matrix}$$

Note that v cannot be among $x_1 \ldots x_p$, by the scoping restriction on the $\forall$-Introduction rule, and so is not affected by the substitution of $\vec{n}$ for $\vec{x}$. Thus

$$\dfrac{\begin{matrix} \mathcal{D}_0^* \\ A[v] \end{matrix}}{\forall u.\, A[u]}$$

is induction-free and data-closed, and can be normalized by applying δ.

We have four logical inferences that close assumptions.

1. Implication introduction:

$$\mathcal{D} \qquad = \qquad \dfrac{\begin{matrix} A_0 \quad B_1 \quad \cdots \quad B_q \\ \mathcal{D}_0 \\ A_1 \end{matrix}}{A_0 \rightarrow A_1}$$

   Since both A_0 and $A_0 \rightarrow A_1$ are data-non-negative, it follows that A_0 is not data-positive, and cannot be one of the B_i's. Thus we may proceed as above.

2. Classical falsehood:

$$\mathcal{D} \qquad = \qquad \dfrac{\begin{matrix} \neg A \quad B_1 \quad \cdots \quad B_q \\ \mathcal{D}_0 \\ \bot \end{matrix}}{A}$$

   Since we did not include contradiction-assumptions among the assumptions B_i, we may again proceed as before.

3. Disjunction elimination:

$$\mathcal{D} \qquad = \qquad \dfrac{\begin{matrix} \vec{B} \\ \mathcal{D}_\vee \\ F_0 \vee F_1 \end{matrix} \qquad \begin{matrix} F_0 \quad \vec{B} \\ \mathcal{D}_0 \\ A \end{matrix} \qquad \begin{matrix} F_1 \quad \vec{B} \\ \mathcal{D}_1 \\ A \end{matrix}}{A}$$

   If $F_0 \vee F_1$ is not data-positive, then neither are F_0 and F_1, and are therefore not among the B_i's. We may thus proceed as in the previous cases.

Suppose then that $F_0 \vee F_1$ is data-positive. By induction assumption we obtain primitive-recursively from input $\vec{n}, \mathcal{C}_1, \ldots \mathcal{C}_q$ a simply-normal data-closed derivation $\begin{array}{c}\mathcal{D}^*_\vee\\ F_0[\vec{n}] \vee F_1[\vec{n}]\end{array}$, equivalent to

$$\mathcal{D}'_\vee \quad = \quad \begin{array}{c} \mathcal{C}_1 \qquad \mathcal{C}_q \\ B_1 \quad \cdots \quad B_q \\ \mathcal{D}_\vee[\vec{n}] \\ F_0[\vec{n}] \vee F_1[\vec{n}] \end{array}$$

Consider the derivation

$$\dfrac{\begin{array}{c} \mathcal{D}^*_\vee \\ F_0[\vec{n}] \vee F_1[\vec{n}] \end{array} \quad \begin{array}{c} F_0[\vec{n}] \quad \vec{B}[\vec{n}] \\ \mathcal{D}_0[\vec{n}] \\ A[\vec{n}] \end{array} \quad \begin{array}{c} F_1[\vec{n}] \quad \vec{B}[\vec{n}] \\ \mathcal{D}_1[\vec{n}] \\ A[\vec{n}] \end{array}}{A[\vec{n}]}$$

By the subformula property of simply-normal derivations, if $F_0 \vee F_1$ were derived in $\mathcal{D}^*_\vee$ by an elimination rule, then it would be a positive subformula of an open assumption of $\mathcal{D}^*_\vee$, contradicting the fact that $\mathcal{D}^*_\vee$ is data-closed. Thus the main inference of $\mathcal{D}^*_\vee$ must be either a Classical Falsehood or a Disjunction Introduction.

In the former case, we have

$$\mathcal{D}^*_\vee \quad = \quad \dfrac{\begin{array}{c} \neg(F_0[\vec{n}] \vee F_1[\vec{n}]) \\ \mathcal{D}^*_{\vee 0} \\ \bot \end{array}}{F_0[\vec{n}] \vee F_1[\vec{n}]}$$

Then the derivation

$$\mathcal{D}^{**}_\vee \quad = \quad \dfrac{\begin{array}{c} \neg(F_0[\vec{n}] \\ \hline\hline \neg(F_0[\vec{n}] \vee F_1[\vec{n}]) \\ \mathcal{D}^*_{\vee 0} \\ \bot \end{array}}{F_0[\vec{n}]}$$

is also data-closed and induction-free. Applying induction assumption to $\mathcal{D}_0$, and using $\xi_{\mathcal{D}_0}$ we obtain a simply-normal data-closed derivation, equivalent to

$$\begin{array}{c} \mathcal{D}^{**}_\vee \quad \mathcal{C}_1 \qquad\quad \mathcal{C}_q \\ F_0[\vec{n}] \quad B_1[\vec{n}] \quad \cdots \quad B_q[\vec{n}] \\ \mathcal{D}_0[\vec{n}] \\ A[\vec{n}] \end{array}$$

If the main inference of $\mathcal{D}^*_\vee$ is Disjunction-Introduction, say

$$\mathcal{D}^*_\vee \quad = \quad \dfrac{\begin{array}{c} \mathcal{D}^*_{\vee 0} \\ F_0[\vec{n}] \end{array}}{F_0[\vec{n}] \vee F_1[\vec{n}]}$$

then we again apply induction assumption to $\mathcal{D}_0$, and using $\xi_{\mathcal{D}_0}$ obtain a simply-normal data-closed derivation, equivalent to

$$\begin{array}{c} \mathcal{D}^*_\vee \quad \mathcal{C}_1 \qquad\quad \mathcal{C}_q \\ F_0[\vec{n}] \quad B_1[\vec{n}] \quad \cdots \quad B_q[\vec{n}] \\ \mathcal{D}_0[\vec{n}] \\ A[\vec{n}] \end{array}$$

4. The case for Existential-Elimination is similar.

We are left with the case of Data-Elimination (i.e. Induction), with

$$\mathcal{D} \quad = \quad \frac{\begin{array}{ccc} \vec{B}[\vec{x}] & \vec{B}[\vec{x}] & A[u,\vec{x}] \quad \vec{B}[\vec{x}] \\ \mathcal{D}_N[\vec{x}] & \mathcal{D}_0[\vec{x}] & \mathcal{D}_s[u,\vec{x}] \\ \mathbf{N}(t[\vec{x}]) & A[\mathbf{0},\vec{x}] & A[\mathbf{s}u,\vec{x}] \end{array}}{A[t[\vec{x}],\vec{x}]}$$

By induction assumption, we obtain primitive recursively in $\vec{\mathcal{C}}$ as above, a simply-normal data-closed derivation

$$\begin{array}{c} \mathcal{D}_N^* \\ \dfrac{\mathbf{N}(t[\vec{n}])}{A[t[\vec{n}],\vec{n}]} \end{array}$$

equivalent to

$$\begin{array}{c} \vec{\mathcal{C}} \\ \vec{B}[\vec{n}] \\ \mathcal{D}_N[\vec{n}] \\ \dfrac{\mathbf{N}(t[\vec{n}])}{A[t[\vec{n},\vec{n}]} \end{array}$$

as well as a simply-normal data-closed derivation $\overset{\mathcal{D}_0^*}{A[\mathbf{0},\vec{n}]}$ equivalent to

$$\begin{array}{c} \vec{\mathcal{C}} \\ \vec{B}[\vec{n}] \\ \mathcal{D}_0 \\ A[\mathbf{0},\vec{n}] \end{array}$$

If A is not data-positive, we also get a simply-normal data-clsoed derivation

$$\begin{array}{c} A[u] \\ \mathcal{D}_s^*[u] \\ A[\mathbf{s}u,\vec{n}] \end{array}$$

equivalent to

$$\begin{array}{c} \qquad \vec{\mathcal{C}} \\ A[u] \quad \vec{B}[\vec{n}] \\ \mathcal{D}_s[u,\vec{n}] \\ A[\mathbf{s}u,\vec{n}] \end{array}$$

We then have, by simple normalization of induction-free derivations, a primitive recursive mapping from data-closed induction-free derivations $\overset{\mathcal{E}}{A[\bar{n}]}$ to a simply-normal data-closed derivation equivalent to

$$\mathcal{D}_s^{*\mathcal{E}} \quad = \quad \begin{array}{c} \mathcal{E} \qquad \vec{\mathcal{C}} \\ A[\bar{n}_0] \quad \vec{B}[\vec{n}] \\ \mathcal{D}_s[\bar{n}_0,\vec{n}] \\ A[\mathbf{s}\bar{n}_0,\vec{n}] \end{array}$$

On the other hand, if A *is* data-positive, then we have, by induction assumption, primitive recursively in data-closed induction-free derivations $\overset{\mathcal{E}}{A[\bar{n}_0]}$ a simply-normal data-closed derivation equivalent to $\mathcal{D}_s^{*\mathcal{E}}$ as above. Note that we may use induction assumption for $\mathcal{D}_s$, because u may be renamed to be different from all x_i's, since no open assumption of $\mathcal{D}_s$ may have u free.

In either case, the premise of Lemma 22 is satisfied, and we obtain primitive-recursively a simply-normal data-closed derivation equivalent to $\mathcal{D}'$. ⊣

5.2 From induction-free proofs to primitive recursion

LEMMA **24** *Suppose that $\mathcal{D}$ is a normal data-closed induction free proof of $\mathbf{N}(t)$ in $\overline{\mathbf{IT}}^+(\mathbb{N}) + \bar{P}$. If n is the number of Successor-Introductions in $\mathcal{D}$, then $P \vdash t = \bar{n}$.*

Proof. By induction on $\mathcal{D}$. By Lemma 21 the main inference of $\mathcal{D}$ is a data introduction or a substitution. If $\mathcal{D}$ is the singleton derivation $\mathbf{N}(\mathbf{0})$ (by Zero-Introduction), then the lemma is trivial. If the main inference is Successor-Introduction,

$$\mathcal{D} \quad = \quad \dfrac{\begin{matrix}\mathcal{D}_0\\ \mathbf{N}(t')\end{matrix}}{\mathbf{N}(\mathbf{s}t')}$$

with t being $\mathbf{s}t'$, then we have, by induction assumption, $P \vdash t' = \bar{m}$, where m is the number of successor introductions in $\mathcal{D}_0$. Thus $P \vdash \mathbf{s}t' = \mathbf{s}\bar{m}$, i.e. $P \vdash t = \bar{n}$ where $n = m+1 =$ the number of successor introductions in $\mathcal{D}$.

Finally, suppose that the main inference of $\mathcal{D}$ is a substitution:

$$\mathcal{D} \quad = \quad \dfrac{\begin{matrix}\mathcal{D}_N & \mathcal{D}_=\\ \mathbf{N}(t') & t' = t\end{matrix}}{\mathbf{N}(t)}$$

Since $\mathcal{D}_=$ is a simply-normal derivation of an equational formula from the equational formulas $\bar{P}$, it is purely equational, thus establishing $P \vdash t = t'$. Also, the number n of successor introductions in $\mathcal{D}$ is the same as the number of successor introductions in $\mathcal{D}_N$. By induction assumption we have $P \vdash t' = \bar{n}$. So we have $P \vdash t = \bar{n}$. ⊣

PROPOSITION **25** *If a function f over $\mathbb{N}$ is provable in $\overline{\mathbf{IT}}^+(\mathbb{N})$, then f is primitive recursive.*

Proof. Assume, without loss of generality, that f is unary. By assumption f is computed by some program $(P, \mathbf{f})$, for which

$$\overline{\mathbf{IT}}(\mathbb{N}), \bar{P} \vdash \mathbf{N}(x) \rightarrow \mathbf{N}(\mathbf{f}(x)).$$

Let $\mathcal{D}$ be a normal derivation in $\overline{\mathbf{IT}}^+(\mathbb{N})$ of $\mathbf{N}(\mathbf{f}(x))$ from assumptions $\bar{P}$ and $\mathbf{N}(x)$. For each $n \in \mathbb{N}$, let $\mathcal{C}_n$ be the direct derivation of $\mathbf{N}(\bar{n})$ using the data-introduction rules. By Lemma 23 there is a primitive recursive function that maps every n and $\mathcal{C}_n$ to a simply-normal data-closed derivation $\mathcal{D}_n^*$ of $\mathbf{N}(\mathbf{f}(\bar{n}))$. By Lemma 24 we have $P \vdash \mathbf{f}(\bar{n}) = \bar{m}$, where m is the number of successor introductions in $\mathcal{D}_n^*$. Thus f is primitive recursive. ⊣

This concludes the proof Theorem 3

References

[1] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 1992.

[2] Stephen Bellantoni and Martin Hofmann. A new feasible arithmetic. *Journal for Symbolic Logic*, 2001.

[3] Stephen J. Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104 (1-3):17–30, 2000.

[4] Samuel Buss. *Bounded Arithmetic.* Bibliopolis, Naples, 1986.

[5] Kurt Gödel. Über eine bisher noch nicht benutzte erweiterung des finiten standpunktes. *Dialectica*, 12:280–287, 1958.

[6] Petr Hájek and Pavel Pudlák. *Metamathematics of First-Order Arithmetic.* Springer, Berlin, 1993.

[7] J.van Heijenoort. *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931.* Harvard University Press, Cambridge, MA, 1967.

[8] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Proceedings of CSL'97*, pages 275–294. Springer-Verlag LNCS 1414, 1998.

[9] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of LICS'99*, pages 464–473. IEEE Computer Society, 1999.

[10] Martin Hofmann. Safe recursion with higher types and bck-algebra. *Annals of Pure and Applied Logic*, 104 (1-3):113–166, 2000.

[11] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, New York, 1980. Preliminary manuscript: 1969.

[12] Daniel Leivant. Substructural proofs and feasibility. Postscript source: www.cs.indiana.edu/ leivant/papers.

[13] Daniel Leivant. Reasoning about functional programs and complexity classes associated with type disciplines. In *Proceedings of the Twenty Fourth Annual Symposium on the Foundations of Computer Science*, pages 460–469, Washington, 1983. IEEE Computer Society.

[14] Daniel Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 279–327. Academic Press, London, 1990.

[15] Daniel Leivant. Subrecursion and lambda representation over free algebras. In Samuel Buss and Philip Scott, editors, *Feasible Mathematics*, Perspectives in Computer Science, pages 281–291. Birkhauser-Boston, New York, 1990.

[16] Daniel Leivant. A foundational delineation of poly-time. *Information and Computation*, 110:391–420, 1994. (Special issue of selected papers from LICS'91, edited by G. Kahn). Preminary report: A foundational delineation of computational feasibility, in Proceedings of the Sixth IEEE Conference on Logic in Computer Science, IEEE Computer Society Press, 1991.

[17] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, Perspectives in Computer Science, pages 320–343. Birkhauser-Boston, New York, 1994. Postscript source: www.cs.indiana.edu/ leivant/papers.

[18] Daniel Leivant. Intrinsic theories and computational complexity. In D. Leivant, editor, *Logic and Computational Complexity*, LNCS, pages 177–194, Berlin, 1995. Springer-Verlag.

[19] Daniel Leivant. Applicative control and computational complexity. In J. Flum and M. Rodriguez-Artalejo, editors, *Computer Science Logic (Proceedings of the Thirteenth CSL Conference*, pages 82–95, Berlin, 1999. Springer Verlag (LNCS 1683). Postscript source: www.cs.indiana.edu/ leivant/papers.

[20] Daniel Leivant. Intrinsic reasoning about functional programs I: first order theories. *Annals of Pure and Applied Logic*, 2001. Postscript source: www.cs.indiana.edu/ leivant/papers.

[21] Charles Parsons. On a number-theoretic choice schema and its relation to induction. In A. Kino, J. Myhill, and R. Vesley, editors, *Intuitionism and Proof Theory*, pages 459–473. North-Holland, Amsterdam, 1970.

[22] D. Prawitz. *Natural Deduction.* Almqvist and Wiksell, Uppsala, 1965.

[23] H.E. Rose. *Subrecursion.* Clarendon Press (Oxford University Press), Oxford, 1984.

[24] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924. English translation: On the building blocks of mathematical logic, in [7], 355-366.

[25] Helmut Schwichtenberg. Feasible programs from proofs. 2000.

[26] Harold Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.

[27] S. Simpson. *Subsystems of Second-Order Arithmetic.* Springer-Verlag, Berlin, 1999.

[28] A.S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, Cambridge, 1996, 2000.

[29] Kai F. Wehmeier. Fragments of ha based on Σ_1-induction. *Archive for Mathematical Logic*, 37:37–49, 1997.

THE BOUNDEDNESS PRINCIPLE

Preliminary Summary

May 2001

Daniel Leivant[*]

1 The boundedness principle and the unity of implicit computational complexity

1.1 Limiting size versus limiting abstraction

Implicit computational complexity characterizes computational complexity classes through rather different approaches, such as function algebras, recurrence schemas, typed applicative calculi, descriptive characterizations in finite model theory, and provable totality in formal theories. Across this diversity is a dichotomy: some characterizations use explicit bounds, i.e. limits on data size, and others use limits on conceptual abstraction. Bounding is central, for instance, to Cobham's bounded recurrence, Buss's Bounded arithmetic, and the global predicates and global functions of finite model theory (more on that below). In contrast, limits on conceptual abstraction are manifested in data ramification (in recurrence or induction), linear logic, and weak comprehension principles in second order logic.

Most of the recent work has focused on limited abstractions, for both theoretical and practical reasons. Conceptually, limiting abstraction is closely related to foundational issues, such as strong forms of predicativity, whereas boundedness is really a machine-level measure in disguise. Practically, conceptual abstractions are promising as a methodology for automatic inference of the computational complexity properties of programs and proofs.

But what is the relation between these two approaches? Can one be explained in terms of the other? If so, we should expect that the more concrete of the two, i.e. boundedness, be explained in terms of the more

[*]Computer Science Department, Indiana University, Bloomington, IN 47405. leivant@cs.indiana.edu. Research partially supported by NSF grant DMS-9870320

foundational, i.e. limited abstraction. For instance, can Cobham's bounded recurrence be justified directly by ramified recurrence, as defined in [5]? Here bounded recurrence for a word algebra $\mathbb{A}(C)$ (where C is the set of constructors) is the definitional template

$$\begin{array}{rcll} \mathbf{f}(\mathbf{c},\vec{x}) & = & \mathbf{g}_c(\vec{x}) & \mathbf{c} \text{ a constant of } C \\ \mathbf{f}(\mathbf{s}w,\vec{x}) & = & \mathbf{g}_s(\vec{x},w,\mathbf{f}(w,\vec{x})) & \mathbf{s} \text{ a unary function identifier of } C \\ |\mathbf{f}(w,\vec{x})| & \leqslant & |\mathbf{B}(w,\vec{x})| & \end{array}$$

Put differently,

$$\begin{array}{rcll} \mathbf{f}(\mathbf{c},\vec{x}) & = & \mathbf{g}_c(\vec{x}) & \\ \mathbf{f}(\mathbf{s}w,\vec{x}) & = & \mathbf{g}_s(\vec{x},w,\mathbf{f}(w,\vec{x})) & \text{truncated to length } |\mathbf{B}(w,\vec{x})| \end{array}$$

Indeed, we'll show momentarily that functions defined by ramified recurrence are closed under bounded recurrence. Underlying the proof is the following observation, which is a simplified form of [5, Lemma 4.1].

LEMMA 1 *Let $\mathbb{A}(C)$ be a word algebra. For each $i \geqslant j \geqslant 0$ there is an* upward-reconstruction function $U_{i,j} : \mathbb{A}_i, \mathbb{A}_j \rightarrow \mathbb{A}_i$, *such that $U_{i,j}(u,v)$ is v in tier i, whenever $|u| \geqslant |v|$.*

Lemma 1 can be construed as stating that one need not worry about ramification of recurrence if all one's operations are is capped by some bound u of higher tier. It seems that the phrase *Boundedness Principle* befits this observation. Interestingly, we have an analogous situation in set theory. The fundamental antinomies of naive set theories, such as Russell's Paradox, are manifestly related to circularity, not to size of sets. Yet the most widely acceptable solution to that problem relies on a doctrine of size: Zermelo replaced the unrestricted comprehension principle by a Separation Schema (Aussonderungsaxiom), stating that one need not worry about comprehension if it is used within an already accepted set.

1.2 Bounded recurrence

We recall some terminology from [6]. The $\mathbf{g}_c$'s and $\mathbf{g}_s$'s functions in the recurrence template above are the *recurrence functions.* For each function $\mathbf{g}_s$ in the template above, the arguments instantiated to $\mathbf{f}(w,\vec{x})$ are the *critical arguments.* If $f : \mathbb{A}_{j_1}, \ldots, \mathbb{A}_{j_k} \rightarrow \mathbb{A}_\ell$, and $d > 0$, then the *d-shift* of f is the copy of f of type $\mathbb{A}_{j_1+d}, \ldots, \mathbb{A}_{j_k+d} \rightarrow \mathbb{A}_{\ell+d}$. Notice that all classes of functions defined by ramified recurrence are closed under all d-shifts.

The following observation is trivial.

LEMMA **2** *For each $i \geqslant j \geqslant 0$ there is a* downward-coercion function $D_{i,j} : \mathbb{A}_i \to \mathbb{A}_j$*, defined by ramified recurrence, and mapping a canonical term a in $\mathbb{A}_i$ to a in $\mathbb{A}_j$.*

LEMMA **3** *Let $\mathbb{A}$ be a word algebra. Given a ramified function $\mathbf{B} : \mathbb{A}_i, \mathbb{A}_j \to \mathbb{A}_{k+1}$, there is a ramified function $\mathbf{B}' : \mathbb{A}_m, \mathbb{A}_j \to \mathbb{A}_k$, ($m = \max(i, k+1)$), defined by ramified recurrence from $\mathbf{B}$, and such that*

1. $|\mathbf{B}'(\mathsf{s}w, x)| \geqslant |\mathbf{B}'(w, x)|$, *and*
2. $|\mathbf{B}'(w, x)| \geqslant |\mathbf{B}(w, x)|$

Proof. Define

$$\begin{array}{rcll} \mathbf{B}'(\mathsf{c}, x) & = & \mathbf{B}(\mathsf{c}, x) & \text{for each constant } c; \\ \mathbf{B}'(\mathsf{s}w, x) & = & \mathbf{B}(\mathsf{s}D_{m,i}w, x) \oplus \mathbf{B}'(w, x) & \text{for each successor } s \end{array}$$

Here $\oplus : \mathbb{A}_{k+1}, \mathbb{A}_k \to \mathbb{A}_k$ is a tiered version of the concatenation function, used in infix notation. ⊣

THEOREM **4** *Let $\mathcal{F}$ be a set of ramified functions over an inductive word algebra $\mathbb{A}(C)$, closed under tier-shifts. If a function f over $\mathbb{A}(C)$ is defined from the (un-tiered version of) functions in $\mathcal{F}$ by bounded recurrence, then it is definable from $\mathcal{F}$ by ramified recurrence and composition.*

Proof. Let f be defined as above, where the recurrence functions and the bounding function are (un-tiered version of functions) in $\mathcal{F}$. To avoid cluttered notation, suppose that $\vec{x}$ consists of a single variable x. Since $\mathcal{F}$ is closed under tier-shift, we may assume that the recurrence functions have a common output tier, t say. Let k be the highest tier of the critical arguments of the recurrence functions. Using again closure under tier-shifts, we may assume that $\mathbf{B}$ has output-tier $k{+}1$. Moreover, by Lemma 3 we may assume that $\mathbf{B}$ is length-increasing with respect to the first argument. Let ℓ be the largest tier assigned to x in the latest ramified versions of the recurrence functions and of $\mathbf{B}$, and let r_0 be the largest tier of w in these ramified functions. Let $r =_{\text{df}} \max(r_0, k{+}1)$.

Define functions $\mathbf{g}'_c : \mathbb{A}_\ell \to \mathbb{A}_t$ (c a constant of C)m $\mathbf{g}'_s : \mathbb{A}_\ell, \mathbb{A}_r, \mathbb{A}_k, \mathbb{A}_{k+1} \to \mathbb{A}_t$ (s a function identifier of C), by

$$\begin{array}{rcll} \mathbf{g}'_c(x) & =_{\text{df}} & \mathbf{g}_c(D_{\ell,i}(x)) & \text{where } i \text{ is the tier of } x \text{ in } \mathbf{g}_c \\ \mathbf{g}'_s(x, w, z, y) & =_{\text{df}} & \mathbf{g}_s(D_{\ell,i}(x), D_{r,j}(w), U_{k+1,k}(y, z))) & \text{where } i, j \text{ are the tiers of } x \text{ and } w \text{ in } \mathbf{g}_s \end{array}$$

Now define the function $f' : \mathbb{A}_r, \mathbb{A}_\ell, \mathbb{A}_{k+1} \to \mathbb{A}_t$ by the ramified recurrence

$$\begin{array}{rcl} \mathbf{f}'(\mathbf{c}, x, y) & = & \mathbf{g}'_c(x) \\ \mathbf{f}'(\mathbf{s}w, x, y) & = & \mathbf{g}'_s(x, w, \mathbf{f}'(w, x, y), y) \end{array}$$

Since $r =$ the tier of the recurrence argument is $> k =$ the tier of the critical arguments, this is a correct ramified recurrence. Now define by ramified composition

$$\mathbf{f}(w, x) = \mathbf{f}'(w, x, \mathbf{B}(w, x))$$

By induction on $|w|$ one verifies that this yields a definition of f, using the facts that j majorizes f and is size-increasing with respect to its first argument. $\dashv$

1.3 Bounded induction

Analogously to the simulation of bounded recurrence by ramified recurrence, Ramified Intrinsic Theories (as defined in [6] justify Bounded Arithmetic). This can be stated for the proof rules, as well as for the definable functions. For instance, we have

THEOREM **5** *The schema $\mathbf{\Sigma}^b_1$-PIND of the system S^1_2 of bounded arithmetic is derivable in the ramified intrinsic theory for binary words, with induction restricted to data-predicative formulas.*[1]

THEOREM **6** *If a function f over binary words is $\mathbf{\Sigma}^b_1$ definable in Buss's S^1_2, then f is provable in the ramified intrinsic theory above.*

More general statements, as well as proofs, are postponed to the full version of this paper.

1.4 Global predicates of finite model theory

If $\mathcal{C}$ is a class of finite structures over a common vocabulary, then a *global r-ary predicate* over $\mathcal{C}$ is a function that assigns to each structure $\mathcal{S} \in \mathcal{C}$ an r-ary relation over the universe $|\mathcal{S}|$ of $\mathcal{S}$. A global predicate can be viewed as

[1]See [6] for the definition of the latter. Note that data-predicative formulas are the intrinsic-theory analogues of the $\mathbf{\Sigma}^0_1$ formulas of arithmetic. Also, recall that S^1_2 is a theory of binary words disguised as a number theory.

a *query* over the structures in $\mathcal{C}$, considered as data-bases. Global predicates can be defined by descriptive devices, such as formulas, or computational devices, such as programs in an imperative or declarative style. Descriptive computational complexity, initiated by Immerman, relates such devices to the complexity of computing the global predicates that they define [4]. In most all cases the structures considered are assumed given with an order.

Global predicates over ordered finite structures can also be viewed as binary-valued functions over binary words. If $\mathcal{S}$ is a structure with universe $\{0, 1, \ldots, n-1\}$, then the given unary relations of $\mathcal{S}$ are codified by their characteristic functions, which are 0-1 words of length n. More generally, r-ary relations are codified as a words that list the entries of r-dimensional matrices of size n, e.g. by using auxiliary character to separate dimensions.[2] While a direct computation over $\mathcal{S}$ yields the truth value of a relation for given arguments as a single step, the simulation would require a polynomial number of steps, with the degree of the polynomial being the arity of the relation. This yields a simulation of Descriptive Complexity Theory by bounded recurrence over words. We give several concrete examples of this change of perspective in the full version of this paper.

The simulation of global predicate by version of bounded recursion is the final chapter in the use of the boundedness principle as a unifying thread in implicit computational complexity.

2 The boundedness principle in use

2.1 Non-size-increasing functions

Much has been said in recent years about the limitation of ramified recurrence to permit common poly-time algorithms. We argue that to a large degree these shortcomings are related to inessential constraints. In particular, the behavior of non-size-increasing functions is related to the boundedness principle.

A typical example is the insertion sort algorithm, for lists over a data

[2]For instance, a binary relation is coded by its 2 dimensional 0-1 matrix, coded as a word in $\{0, 1, \$\}^*$, with \$ used to separate the listings of rows.

type $\mathbb{A}$ with an order relation $<$:[3]

$$\begin{array}{rcl}
\text{insert}(a, []) & = & [a] \\
\text{insert}(a, b :: l) & = & \text{if } a \leqslant b \\
 & & \quad \text{then } a :: (b :: l) \\
 & & \quad \text{else } b :: \text{insert}(a, l) \\[4pt]
\text{sort}([]) & = & [] \\
\text{sort}(a :: l) & = & \text{insert}(a, \text{sort}(l))
\end{array}$$

The definition of insert can be trivially ramified, but with the input-list at higher tier than the output tier. This blocks the ramification of sort. The algorithm is nonetheless in poly-time, because insert does not increase the combined size of its input. Hofmann has developed a type system to address this and similar issues [3]. Among its several innovative features, it deals with the issue of size-increase by typing the constructors not as functions from objects to objects, but as functions that also consume a singleton type $\diamond$, a "construction permit token" so to speak.

Hofmann's formalism is highly innovative on many counts, and opens new possibilities for automatic inference of type systems that guarantee complexity bounds. The complexity of his type system is then relatively inconsequential, since it is transparent to the programmer. The point we raise here is that from a purely mathematical viewpoint, non-size-increasing functions have a straightforward treatment in the framework of ramified recurrence, albeit not one that lends itself to easy type inference. Indeed, non-size-increasing functions can be defined by free use of recurrence, simply by using the input itself as the bound. In the particular example above, one might even use simply the list input of insert as a "clock", that is, defining *insert* : $\mathbb{A}_0, \mathbb{L}_0, \mathbb{L}_1 \to \mathbb{L}_0$, where $\mathbb{L}$ is the type of $\mathbb{A}$-lists. The last list argument is the clock:

$$\begin{array}{rcl}
\text{insert}(a, [], m) & = & [a] \\
\text{insert}(a, b :: l, c :: m) & = & \text{if } a \leqslant b \\
 & & \quad \text{then } a :: (b :: l) \\
 & & \quad \text{else } b :: \text{insert}(a, l, m)
\end{array}$$

[3]We follow [3] in using [] for the empty list, and $a :: l$ for cons(a, l).

We can now define $\underline{\text{sort}}$: $\mathbb{L}_1 \rightarrow \mathbb{L}_0$ by ramified recurrence:

$$\begin{aligned}\text{sort}([]) &= [] \\ \text{sort}(a :: l) &= \text{insert}(a, D_{1,0}\text{sort}(l), l)\end{aligned}$$

A caveat is that the definition above of $\underline{\text{insert}}$ is no longer a recurrence, but rather a recurrence with parameter substitution. However, the substitution is performed exactly once, and it has been known for a while (see e.g. [1]) that this form of more liberal ramified recurrence still preserves poly-time. The general form for such recurrences is (for a single sorted inductive algebra $\mathbb{A}(C)$)

$$\begin{aligned}f(\mathbf{c}(x_1 \ldots x_r), \vec{y}, z) &= g_c(\vec{x}, \vec{y}, f_{c1}, \ldots f_{cr}) \\ &\text{where} \\ f_{ci} &= f(x_i, \vec{y}, h_{ci}(\vec{y}, z)) \\ &\qquad \text{for each constructor } \mathbf{c} \text{ of } C\text{, where } r = \underline{arity}(\mathbf{c}) \geqslant 0\end{aligned}$$

In the ramified version we require that the recurrence argument be of tier higher than the critical arguments. Note that in the parameterizing functions h_{ci} the argument z has the same tier as that of the function's output, so such functions, if defined by ramified recurrence, are explicitly defined from the algebra constructors.

The ramified schema above preserves the function feasibility, in the following sense: if all parameterizing functions are computable in constant time on a pointer machine for $\mathbb{A}(C)$, then f is computable in poly-time on such machines from the functions g_c. If the algebra $\mathbb{A}(C)$ has constructors of arity $\geqslant 2$, then pointer machines over $\mathbb{A}(C)$ are not in general simulated by Turing machine. Thus, we obtain truly poly-time complexity only when the recurrence argument has only one predecessor.

2.2 Singly-parameterized recurrence in multi-sorted data-systems

Consider now multi-sorted data-systems, as described in [6]. The general form of recurrence requires in general simultaneous recurrence for all the sorts, since those may be generated simultaneously. However, if the sorts are generated hierarchically, then there is no need for simultaneous recurrence. For example, in the data system consisting of a sort $\mathbb{W}$ of binary words and a sort $\mathbb{L}$ of lists over $\mathbb{W}$, then the generation of $\mathbb{W}$ is completed independently of

$\mathbb{L}$. The list constructor cons, of type $\mathbb{W}, \mathbb{L} \rightarrow \mathbb{L}$, has therefore one destructor, and the corresponding clause of the recurrence template takes the form

$$\begin{array}{rcl} \mathbf{f}(\mathrm{cons}(w,l),\vec{y},z) & = & \mathbf{g}_c(w,l,\vec{y},\mathbf{f}_c) \\ \text{where} & & \\ \mathbf{f}_c & = & \mathbf{f}(l,\vec{y},h_c(\vec{y},z)) \end{array}$$

2.3 Ramified regression

The starting point of our discussion was the presence of fast algorithms, stated as equational recurrences, that cannot be ramified. But the issue goes beyond ramification. Consider the following algorithm for the difference (in absolute value) of two natural numbers:

$$\begin{array}{rcl} \mathrm{diff}(\mathbf{0},x) & = & x \\ \mathrm{diff}(x,\mathbf{0}) & = & x \\ \mathrm{diff}(\mathbf{s}x,\mathbf{s}y) & = & \mathrm{diff}(x,y). \end{array}$$

From Colson's work [2] we know that diff has no primitive recursive definition that runs in time proportional to the smallest of the inputs, a property that is evident for the definition above.

It seems, therefore, that we should be interested not only in ramification/typing systems that admit more definition by recurrence, but in new types of recurrence as well, i.e. in the broader question of rewrite systems, and their ramification. As a modest beginning, let us propose a schema of ramified recurrence that admits all examples above. For clarity, we state it first for word algebras $\mathbb{A}(C)$; the generalizations to arbitrary inductive algebras, and further on to multi-sorted inductive data-systems, is unproblematic.

We call the terms generated from variables and the constructors of C *base terms*. For base terms $\mathbf{t}, \mathbf{t}'$ we write $\mathbf{t} \preccurlyeq \mathbf{t}'$ if $\mathbf{t}$ is a subterm of $\mathbf{t}$, and $\mathbf{t} \prec \mathbf{t}'$ if it is a strict subterm of $\mathbf{t}'$. For tuples $\vec{\mathbf{t}} = \langle \mathbf{t}_1 \ldots \mathbf{t}_k \rangle$ and $\vec{\mathbf{t}}' = \langle \mathbf{t}'_1 \ldots \mathbf{t}'_k \rangle$ we write $\vec{\mathbf{t}} \prec \vec{\mathbf{t}}'$ if $\mathbf{t}_i \preccurlyeq \mathbf{t}'_i$ for $i = 1 \ldots k$, and $\mathbf{t}_j \prec \mathbf{t}'_j$ for at least one j.

The schema of *singly-parameterized regression* allows the definition of a partial function f over $\mathbb{A}(C)$ by clauses of the form

$$\begin{array}{rcl} f(\vec{\mathbf{t}}_\ell,z) & = & g_\ell(\vec{x},z,f_\ell) \\ \text{where} & & \\ f_\ell & = & f(\vec{\mathbf{t}}'_\ell,\vec{x},h(\vec{x},z)) \end{array}$$

Here $\vec{x}$ are the variables in $\vec{\mathbf{t}}$. The latter are dubbed the *regression argument* of the clause. The essential requirements are:

1. $\vec{\mathbf{t}}' \prec \vec{\mathbf{t}}$ in each clause.
2. If $\vec{\mathbf{t}}_0$ and $\vec{\mathbf{t}}_1$ are the regression arguments of two clauses, then they cannot be unified. This guarantee that the clauses do not generate multi-valued functions, but we see no reason to exclude the definition of partial functions.

In the ramified version of the schema above we require that the tiers of the regression arguments all exceed the tier of the critical argument f_ℓ; i.e.: if there is a critical argument, then the tiers of the regression arguments all exceed the output-tier.

Note that Colson's algorithm is a simple case of the template above; moreover, it is obviously ramified.

THEOREM **7** *If a function f over an inductively generated algebra $\mathbb{A}(C)$ is defined by ramified singly-parameterized regression, then if is computable in poly-time on a pointer machine over $\mathbb{A}(C)$. In particular, if $\mathbb{A}(C)$ is a word algebra, then f is poly-time.*

THEOREM **8** *If a function f as above is defined by an (un-ramified) singly-parameterized regression, with a bounding function j, then f is defined from the regression functions and j by ramified singly-parameterized regression.*

2.4 Flat versus non-size-increasing functions

Consider again the example above of insertion-sort. We have shown that the recursive definition can be ramified, provided we admit singly-parameterized recurrence. The input list l is used in two ways: as a higher-tier object to clock the computation, and as lower-tier object that is being transformed into another list *of the same low tier.* Put differently, the high-tier copy of l iterates a "flat" function (albeit a function with a high-tier parameter). The fact that insert is non-size-increasing was a related facet of the same situation, but not an essential facet.

To see the difference consider the following program, which when given a list of elements as input, produces a sorted list with each element duplicated.

$$
\begin{aligned}
\text{dupinsert}(a, []) &= [a, a] \\
\text{dupinsert}(a, b :: l) &= \text{if } a \leqslant b \\
&\qquad \text{then } a :: a :: b :: l \\
&\qquad \text{else } b :: \text{dupinsert}(a, l) \\
\text{dupsort}([]) &= [] \\
\text{dupsort}(a :: l) &= \text{dupinsert}(a, \text{dupsort}(l))
\end{aligned}
$$

Here dupsort is not length-preserving, and yet the definition can be ramified just like the definition of sort above.

References

[1] A. Beckmann and A. Weiermann. A term rewriting characterization of the polytime functions and related complexity classes. *Archive for Mathematical Logic*, 36:11–30, 1996.

[2] Loic Colson. About primitive recursive algorithms. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *Proceedings of the 16th International Colloquium on Automata, Languages and Programming, Stresa, Italy*, pages 194–206. Springer-Verlag LNCS 372, July 1989.

[3] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of LICS'99*, pages 464–473. IEEE Computer Society, 1999.

[4] Neil Immerman. *Descriptive Complexity.* Springer, Berlin, 1999.

[5] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, Perspectives in Computer Science, pages 320–343. Birkhauser-Boston, New York, 1994.

[6] Daniel Leivant. Intrinsic reasoning about functional programs i: first order theories. *Annals of Pure and Applied Logic*, 2001.

SUBSTRUCTURAL TERMINATION PROOFS AND FEASIBILITY CERTIFICATION

Extended Abstract

May 2001

Daniel Leivant*

Abstract

We refer to termination proofs for equational programs, in the framework of [26] for verification of equational programs, dubbed *intrinsic theories.* We show that a natural notion of data ramification yields variants of intrinsic theories, whose provably total functions are precisely poly-time, linear-space, or Kalmar-elementary, depending on the underlying data and the allowable instances of induction. Using an orthogonal approach, we showed in [27] that natural structural conditions on the use of induction lead to restricted intrinsic theories whose provably total functions are precisely major computational complexity classes: depending on the theories, the classes are poly-time, linear-space, or primitive recursive. All these theories provide, therefore, a setting for automatic inference of program complexity, without explicit reference to implementation, machine models, or dataflow analysis.

In those intrinsic theories above that characterize poly-time, linear-space, or primitive recursion, induction is limited to "data non-negative formulas" (a generalization of $\mathbf{\Sigma}^0_1$ formulas). When induction is permissible for all formulas, additional functions become provable: the formalisms for poly-time and linear-space yield then the Kalmar-elementary functions, and those for primitive recursion the provably recursive functions of Peano Arithmetic. Here we also consider these intrinsic theories, with induction permissible for all formulas in the language, but where the multiplicity of working assumptions (in a natural deduction) is restricted, at least for non-data-positive formulas. We show that when multiplicity is disallowed, unrestricted induction does not yield any new provably total functions: we still get poly-time and linear-space, as for the original intrinsic theories.

Key Words: Implicit computational complexity, proof theory, substructural proofs, intrinsic theories, program verification, ramified induction, equational programs, program termination, feasibility, polynomial time, linear space, elementary functions, typed lambda calculi.

*Computer Science Department, Indiana University, Bloomington, IN 47405. `leivant@cs.indiana.edu`. Research partially supported by NSF grant DMS-9870320

1 INTRODUCTION

1.1 Implicit computational complexity

This paper is a contribution to Implicit Computational Complexity, i.e. the collection of approaches to computational complexity that have emerged over recent years, which define and classify the complexity of computations without direct reference to an underlying machine model. The motivation is that the complexity of a computation should be visible from its high level specification, without a machine level examination of resources, such as time and space. The machine-independent high-level approaches for characterizing computational complexity cover a wide range, including applicative functional programming languages, linear logic, bounded arithmetic and bounded set theory, database languages interpreted over finite structures, and structural restrictions on program termination proofs. Each such approach introduces measures of resources and corresponding notions of complexity, such as complexity of proofs, kinds of set existence principles, numbers and order of variables, etc. Close correspondences have been unraveled between such approaches and major computational complexity classes, as well among various approaches. These are testimony to the fundamental and robust nature of the concepts being explored. The hope is that the conceptual approach to computational complexity being developed will not only enhance our understanding of difficult questions in complexity but also, through migration of these concepts into database theory, functional programming languages, and formal methods in hardware and software design, aid in the engineering management of complexity.

1.2 Proof theoretic computational complexity: general aims

The proof theoretic approach to implicit computational complexity identifies major functional complexity classes as consisting precisely of the computable functions that are proved to be well defined (i.e. terminate for all input) in particular formal theories. There are two broad application areas for this effort. From the viewpoint of software engineering, one would hope to have formal tools for program verification development that would guarantee feasible execution of those programs verified and generated, without direct reference to the implementation of these programs. On the other hand, from the viewpoint of Feasible Mathematics the aim is to identify proof methods that are "feasibly safe"; that is, where theorems proved using only such methods are guaranteed to be true feasibly; e.g., such that if one proves a formula $\forall x \exists y \varphi[x, y]$ over alphanumeric words, where φ is quantifier free, then there is a poly-time functions f such that $\varphi[x, f(x)]$.

A number of characteristics are beneficial in meeting the two broad aims above. These might not be all completely achievable or compatible, but they are desiderata worth keeping in mind.

1. *Fit with programming languages.* Formal tool for reasoning about programs should mesh well with the programs themselves.

2. *Data genericity.* The natural numbers have been the underlying data of foundational mathematics, whereas the canonical data of computing theory is words over finite alphabets. The two approaches are basically isomorphic within sufficiently powerful tools, where words are coded by numbers. They are quite disjoint in feasible mathematics. Our formal methods should therefore address primarily symbolic data, not natural numbers. Better yet, it might address generically all forms of inductively generated data, including tree algebras (which are useful for fast data-retrieval in implementations of databases).

3. *Transparency.* The formalisms should be compatible with a user friendly formal development of mathematics. The fact that the structure of certain proofs guarantee certain computational properties of the theorems they derive is then to be automatically deduced as an afterthought, so to speak. One may then envision a library of formal mathematics, presented in an un-obstructive style, and where useful computational complexity bounds automatically ensue from a simple observation of the proofs.

4. *Programming generality.* Clearly, the programming language one refers to must be general: it would make no sense, for instance, to study proof theoretically a programming language already known to capture poly-time, say Bellantoni-Cook's safe recursion [2]. More interestingly, one wishes to capture not only all *functions* of complexity class X, but as many *programs/algorithms* in X.

1.3 Contributions of this paper

We refer here to the verification methodology for equational programs of [23, 26], dubbed *intrinsic theories.* The underlying idea is very simple, yet surprisingly fruitful: for each inductively-generated data system C we create a skeletal theory $\mathbf{IT}(C)$, whose axioms are merely data-introduction axioms, i.e. the closure of data under the basic constructors, and data-elimination, i.e. induction axiom-schemas for the data types.

The results reported here are in two directions. First, we define ramified intrinsic theories, for reasoning about general equational programs, and we show that they capture precisely poly-time, linear-space, and Kalmar-elementary, depending on structural restrictions on induction. These formalisms were described in broad strokes in [23], but the results are unpublished. We then investigate options for allowing induction for all formulas, without spilling out of poly-time. The conditions we obtain can be stated either by structural properties of proofs, or, more attractively, by substructural logical rules. These rules bear loose relation to linear logic, but do not invoke modal operators or linear connectives.[1] Similar results hold for the predicative induction of [27], as reported above. These results further the usefulness of intrinsic theories as a framework for transparently deriving computational complexity properties of equational programs, as they allow greater flexibility in permissible proof methods.

[1]The relation to Linear Logic is more tenuous than may first seem.

1.4 Comparison with other works

Proof Theory has a long tradition of investigating relations between provable properties of (computable) functions and their computational complexity. Most significant was Gödel's discovery that the provably recursive functions of first order (Peano) arithmetic are precisely the functions definable by primitive recursion in all finite types [9]. This approach was applied by Buss to computational complexity by focusing attention on formulas in which quantifiers are suitably bounded [6, 16]. Bounded Arithmetic ties well with traditional proof theory and its well-developed tools, but has virtually no fit with any programming language (functions are referred to via codings, as only a few basic functions are explicitly named).

Proof theory for direct, coding-free reasoning about equational programs was introduced in [19], and related to computational complexity in [21]. These studies focused on second order formalizations, with feasibility of provable functions being guaranteed via restrictions on set-existence (i.e., comprehension). First order intrinsic theories were introduced in [23], with the feasible variant using data ramification, a concept considered earlier for function definition by recurrence, and introduced independently in [28], [20], and [2].[2]

Ramified recurrence for higher-type functions leads to functions of exponential growth, by allowing size-duplication at each cycle of an iteration ([17], revised as [25]). Analogously, induction for arbitrary formulas has the same effect on the provable functions of ramified intrinsic theories. Recent studies have attempted to counteract these computational effects of higher-type recursion and unrestricted induction, by using syntactic restrictions that block duplications. One variant of this approach simply refers to an applicative calculus for recurrence, and simply disallows duplications of certain variables; this idea is basically due to Neil Jones [14], and refined in [24].[3] Another variant uses linear types and modalities to achieve a similar effect [5].[4]

A proof theoretic analog to [5] was developed by Bellantoni [4]. This was greatly streamlined in [3], a formalism which does away with the explicit use of linear implication, and uses the modal $\Box$ to both simulate ramification and to permit limited forms of duplications. This work bears substantial similarities to our Theorem 7 (though *not* to the rest of the paper, in particular not Theorem 5, which is of independent interest). However, Theorem 7 is stronger than [5] in at least two important ways. First, we permit multiplicity of assumptions for a much broader class of formulas than [5]. Second, induction in our system is for all formulas, including references to all tiers, whereas induction in [5] is restricted to $\Box$-free formulas, i.e. to formulas that refer to base tier only. Moreover, our entire treatment is generic with respect to the underlying data. Finally, our formalism is more transparent, in the following sense: each of our proofs is a correct proof in the unrestricted intrinsic theory, with tiering information added. A proof in [5] would typically contain extensive uses of axioms and inference rules for the $\Box$ operator; this machinery can be eliminated, yielding an

[2]The use of ramified data as a sorted algebra with well-typed composition was introduced in [22].

[3]This method is combined there not with data ramification, but with an alternative method of controlling unfolding of recurrence.

[4]A modal type constructor $\Box$ is used there to generate tiers, a method introduced independently in [11] and [25], where ρ is used in place of $\Box$. However, it seems that the modality also serves there to prevent duplication, in combination with the linear implication.

unrestricted proof, but it is a far trickier feet to recover the original, modal proof, from the latter "collapse".

2 INTRINSIC REASONING ABOUT EQUATIONAL PROGRAMS

2.1 Intrinsic reasoning about equational programs

In [23, 26] we introduced a verification methodology for equational programs, dubbed *intrinsic theories.* For each inductively-generated data system C one uses a skeletal theory $\mathbf{IT}(C)$, whose axioms are merely data-introduction axioms, i.e. the closure of data under the basic constructors, and data-elimination, i.e. induction axiom-schemas for the data types. We focus here on a single-sorted set C of constructors, so the functionality of each constructor is determined by its arity, a natural number $\geqslant 0$. The constructors of arity 0 are the *constants.* If C has no constant then there are no C-terms, and if it has no constructor of arity > 0 then the set of terms is finite. In either case we say that C is *trivial.* If all constructors in C have arity $\leqslant 1$, and at least two constructors have arity 1, then the generated algebra $\mathbb{A}(C)$ is a *word algebra.* If a non-trivial C has only constants and one constructor of arity 1, then $\mathbb{A}(C)$ is a *unary algebra.*

The vocabulary of the intrinsic theory $\mathbb{A}(C)$ consists of the constructors of C, and one unary relation identifier D (or one for each sort, for multi-sorted data system). Suppose C refers to a single ground type N, and has the constant $\mathbf{0}$ and the unary constructor s, thus generating the set $\mathbb{N}$ of natural number (in unary notation). Writing N for the unary relation identifier D, the axioms of $\mathbf{IT}(\mathbb{N})$ are then the data-introduction axioms, which written as inference rules read

$$\frac{}{\mathsf{N}(\mathbf{0})} \qquad \text{and} \qquad \frac{\mathsf{N}(\mathbf{t})}{\mathsf{N}(\mathbf{st})}$$

and the induction schema (only one for a single-sorted data-system),

$$\frac{\mathsf{N}(t) \quad \varphi[\mathbf{0}] \quad \forall x.\, \varphi[x] \rightarrow \varphi[\mathsf{s}x]}{\varphi[t]}$$

See [26] for the generic rules.

We refer to equational programs over the data-system in hand. Each such program consists of a finite set P of equations between terms, where the terms are built from variables, the constructors ($\mathbf{0}$ and s in the case of $\mathbb{N}$), and program function-identifiers. One identifier, $\mathbf{f}$, is singled out as the program's principal identifier. If $\mathbf{f}$ is r-ary, and f is an r-ary function over the data-system, then we say that P *computes* f when, for all base terms $a_1 \ldots a_r, b \in \mathbb{A}(C)$, $f(a_1, \ldots, a_r) = b$ exactly when the formal equation $\mathbf{f}(a_1, \ldots, a_r) \asymp b$ is derived from P in equational logic. Note that this definition allows arbitrary recursion, including simultaneous recursions.

A program P with principal function identifier $\mathbf{f}$, say a unary function over $\mathbb{N}$, is said to be *provable*, if

$$\mathbf{IT}(C), \forall P, \mathsf{N}(x) \vdash \mathsf{N}(\mathbf{f}(x))$$

where $\forall P$ is the universal closure of P.

THEOREM 1 [23, 26] *The provable functions of* $\mathbf{IT}(\mathbb{N})$ *are precisely the provably-recursive functions of Peano Arithmetic. More generally, if* $\mathbb{A}(C)$ *is a non-trivial inductive algebra, then the provable functions of* $\mathbf{IT}(C)$ *are precisely the functions over* $\mathbb{A}(C)$ *whose numerically-coded counterparts under any of the standard numeric data codings) are provably-recursive in Peano Arithmetic.*

The intrinsic framework is generic with respect to data systems, and it enables explicit reasoning about functional programs without recourse to numeric codes or a logic of partially-denoting terms. Among these programs are ones whose termination cannot be proved within the formal theory used. This makes it possible to freely refer to partial computable functions whose termination cannot be proved; for example, in a formalism for poly-time we can refer to Ackermann's Function, though its termination is surely unprovable. Conceptually, the framework lends itself to a delineation of various forms of finitistic and predicative ontologies of data, and to proof theoretic characterizations of computational complexity classes, as we do here. Such formalisms are quite different from the well developed framework of Bounded Arithmetic, and offer an expressively rich and unobtrusive setting for formalizing Feasible Mathematics, e.g. Poly-time or Poly-space Mathematics. Moreover, the framework can be adapted to other form of declarative programming, such as logic programs. We refer the reader to [26] for additional details, motivation, background, and fundamental proof theoretic results.

2.2 Easy proofs of unfeasible functions

The intrinsic framework makes it clear why certain rapidly growing functions have easy proofs. For reference in the sequel, let us show that exponentiation over $\mathbb{N}$ is provable, by referring to two definitions of exponentiation. First, we have the usual definition by (primitive) recurrence of exponentiation from addition and multiplication.

$$\begin{array}{rcl@{\qquad}rcl@{\qquad}rcl}
x+\mathbf{0} &=& x & x\times\mathbf{0} &=& \mathbf{0} & x\#\mathbf{0} &=& \mathsf{s}\mathbf{0} \\
x+\mathsf{s}y &=& \mathsf{s}(x+y) & x\times\mathsf{s}y &=& (x\times y)+x & x\#\mathsf{s}y &=& (x\#y)\times x
\end{array}$$

For this program we have the following proof, where for readability we omit uses of the program, and use instead double-bars to indicate such uses (via equational rules), as well as other trivial short-cuts.

$$\begin{prooftree}
\AxiomC{$\mathsf{N}(q)$}
\AxiomC{$\mathsf{N}(\mathbf{0})$}
\UnaryInfC{$\mathsf{N}(\mathsf{s}\mathbf{0})$}
\doubleLine
\UnaryInfC{$\mathsf{N}(x\#\mathbf{0})$}
\AxiomC{$\mathsf{N}(x)$}
\AxiomC{$\mathsf{N}(\mathbf{0})$}
\doubleLine
\UnaryInfC{$\mathsf{N}((x\#p)\times\mathbf{0})$}
\AxiomC{$\mathsf{N}(x\#p)$}
\AxiomC{$\mathsf{N}((x\#p)\times z)$}
\UnaryInfC{$\mathsf{N}((x\#p)\times z+\mathbf{0})$}
\AxiomC{$\mathsf{N}((x\#p)\times z+u)$}
\UnaryInfC{$\mathsf{N}(\mathsf{s}((x\#p)\times z+u))$}
\doubleLine
\UnaryInfC{$\mathsf{N}((x\#p)\times z+\mathsf{s}u)$}
\TrinaryInfC{$\mathsf{N}(((x\#p)\times z)+x\#p)$}
\doubleLine
\UnaryInfC{$\mathsf{N}((x\#p)\times\mathsf{s}z)$}
\RightLabel{ind}
\TrinaryInfC{$\mathsf{N}((x\#p)\times x)$}
\doubleLine
\UnaryInfC{$\mathsf{N}(x\#(\mathsf{s}p))$}
\RightLabel{ind}
\TrinaryInfC{$\mathsf{N}(x\#q)$}
\end{prooftree}$$

Note that here the induction assumption $\mathbf{N}(x\#p)$ is used as the major premise of a (nested) induction.

An alternative definition of the base-two exponential function **exp** is

$$\begin{array}{rcl} \mathbf{e}(\mathbf{0},y) & = & \mathbf{s}y \\ \mathbf{e}(\mathbf{s}x,y) & = & \mathbf{e}(x,\mathbf{e}(x,y)) \end{array} \qquad \mathbf{exp}(x) = \mathbf{e}(x,\mathbf{0}).$$

A natural deduction in $\mathbf{IT}(C)$ for this program:

$$\dfrac{\dfrac{\dfrac{\mathbf{N}(x) \quad \dfrac{\dfrac{\dfrac{\mathbf{N}(y)}{\mathbf{N}(\mathbf{s}y)}}{\mathbf{N}(\mathbf{e}(\mathbf{0},y))}}{\forall y\,(\mathbf{N}(y)\rightarrow\mathbf{N}(\mathbf{e}(\mathbf{0},y)))} \quad \dfrac{\dfrac{\dfrac{\dfrac{\forall y\,(\mathbf{N}(y)\rightarrow\mathbf{N}(\mathbf{e}(u,y)))}{\mathbf{N}(\mathbf{e}(u,y))\rightarrow\mathbf{N}(\mathbf{e}(u,\mathbf{e}(u,y)))} \quad \dfrac{\dfrac{\forall y\,(\mathbf{N}(y)\rightarrow\mathbf{N}(\mathbf{e}(u,y)))}{\mathbf{N}(y)\rightarrow\mathbf{N}(\mathbf{e}(u,y))} \quad \mathbf{N}(y)}{\mathbf{N}(\mathbf{e}(u,y))}}{\mathbf{N}(\mathbf{e}(u,\mathbf{e}(u,y)))}}{\mathbf{N}(\mathbf{e}(\mathbf{s}u,y))}}{\forall y\,(\mathbf{N}(y)\rightarrow\mathbf{N}(\mathbf{e}(\mathbf{s}u,y)))}}{\forall y\,(\mathbf{N}(y)\rightarrow\mathbf{N}(\mathbf{e}(x,y)))}\,ind}{\mathbf{N}(\mathbf{e}(x,\mathbf{0}))}}{\mathbf{N}(\mathbf{exp}(x))}$$

Note that here the induction formula has $\mathbf{N}$ in both positive and negative positions. Moreover, the assumption $\forall y\,(\mathbf{N}(y)\rightarrow\mathbf{N}(\mathbf{e}(u,y))$ is used twice.

2.3 Intrinsic theories and computational complexity

Intrinsic theories lend themselves to at least four sorts of restrictions: data ramification, structural conditions on induction formulas, structural conditions on working assumptions of inductions, and conditions on multiplicity of working assumptions.

Structural conditions on induction formulas and assumptions are studied in [27]. Two sorts of restrictions are considered there: limiting the form of induction formulas, and limiting the dependence of induction's data (=leftmost) assumption. As usual, say that an occurrence of $\mathbf{N}$ in a formula A is *positive* (respectively, *negative*) if it is in the negative scope of an even (respectively, odd) number of implications and negations. Call a formula φ in the vocabulary of $\mathbf{IT}(C)$ *data-positive* if $\mathbf{D}$ has positive occurrences in φ, and *data-predicative* if it does not have both positive and negative occurrences of $\mathbf{D}$. For example, the renditions of $\mathbf{\Sigma}^0_1$ and of $\mathbf{\Pi}^0_1$ formulas in $\mathbf{IT}(\mathbb{N})$ are data-predicative. Indeed, a $\mathbf{\Sigma}^0_1$ formula $\exists x\ \varphi_0$, where φ_0 is a quantifier-free formula of primitive-recursive arithmetic, is rendered by $\exists x\ \mathbf{N}(x) \wedge \varphi_0$, in which $\mathbf{N}$ occurs only positively. Similarly, the $\mathbf{\Pi}^0_1$ formula $\forall x\ \varphi_0$ is rendered by $\forall x\ \mathbf{N}(x)\rightarrow\varphi_0$, in which $\mathbf{N}$ occurs only negatively. However, the renditions of $\mathbf{\Sigma}^0_2$ and of $\mathbf{\Pi}^0_2$ has $\mathbf{N}$ occurring both positively and negatively, and are therefore not data-predicate. Note, also, that the induction formula in the proof above for the function **exp** is not data-predicative.

Call an open assumption of the data (=leftmost) premise of induction in a derivation $\mathcal{D}$ a *working induction dependence* if it is closed in $\mathcal{D}$. Call a derivation $\mathcal{D}$ *induction-predicative* if it has no data-positive working induction dependence. Note that this notion refers to the derivation as a whole, and not to the instance of induction on its own. The proof above for the exponential function # is not induction-predicative.

An alternative way of stating predicative induction, without conditions on proofs as a whole, uses the notion of *dual-sequents,* i.e. triplets $\Theta; ,\ \Rightarrow \varphi$, where Θ and , are multi-sets of formulas, and φ is a formula. The intent is that Θ is the set of global assumptions, which are not closed by inferences. An *initial sequent* is then of the form $\Theta; ,\ \Rightarrow \varphi$ where $\varphi \in \Theta \cup ,$. Other than that, natural deduction inferences leave the set of global assumptions unaffected. *Predicative induction,* for $\mathbb{N}$ say, takes then the form:

$$\frac{\Theta; ,_1 \Rightarrow \mathbf{N}(t) \quad \Theta; ,_2 \Rightarrow \varphi[\mathbf{0}] \quad \Theta; ,_3, \varphi[z] \Rightarrow \varphi[\mathbf{s}z]}{\Theta;\ ,_1, ,_2, ,_3 \Rightarrow \varphi[\mathbf{t}]}$$

under the proviso that no formula in $,_1$ is data-positive (and, as usual, z is not free in the derived dual-sequent).

THEOREM **2** [27]

1. *The functions provable in* $\mathbf{IT}(C)$ *with induction restricted to data-predicative formulas are precisely the functions computable in primitive recursive time.*

2. *The functions provable by induction-predicative proofs in* $\mathbf{IT}(C)$ *are precisely the Kalmar-elementary functions.*

3. *The functions provable in* $\mathbf{IT}(C)$ *under both restrictions above, are precisely the functions computable in poly-time on a register machine over* $\mathbb{A}(C)$*, i.e. the poly-time functions in case* $\mathbb{A}(C)$ *is a word algebra, and the linear-space functions in case* $\mathbb{A}(C)$ *is a unary algebra.*

3 Ramified intrinsic theories

3.1 Definition of the formalism

As outlined in the Introduction, we consider single-sorted inductively generated algebras. The simplest non-trivial example is the single-sorted algebra $\mathbb{N}$ of the natural numbers, generated by a zero-ary constructor $\mathbf{0}$ and a unary constructor $\mathbf{s}$ (the successor function). The single-sorted algebra $\mathbb{W}$ of binary words is generated from a zero-ary $\boldsymbol{\varepsilon}$ (the empty word) and two unary functions $\mathbf{0}$ and $\mathbf{1}$; we identify the words ϵ, 0, 1, 00, ... over $\{0, 1\}$ with the $\mathbb{W}$-terms $\boldsymbol{\varepsilon}, \mathbf{0}\boldsymbol{\varepsilon}, \mathbf{1}\boldsymbol{\varepsilon}, \mathbf{00}\boldsymbol{\varepsilon}, \ldots$, generated from the constructors. The general case of a single-sorted system has a finite set $C = \{\mathbf{c}_1 \ldots \mathbf{c}_k\}$ of constructors (i.e. reserved function identifiers) of some arities $r_1 \ldots r_k \geqslant 0$ respectively, from which the *term algebra* $\mathbb{A}(C)$ of the closed terms over $\mathbf{c}_1 \ldots \mathbf{c}_k$ is generated inductively. We shall not dwell here on multi-sorted inductive data systems, which are treated in [27].

The *ramified intrinsic theory* for $\mathbb{A}(C)$, $\mathbf{RIT}(C)$, is the first order theory defined as follows. The *vocabulary* consists of the constructors in C, and of unary relational identifiers D_1, D_1, D_2, $\ldots$, which we dub the *tiers.* The *axioms*, which we write as inference rules, consist of *data introduction axioms* and a *data elimination* (induction) schema. The former state that each D_i is closed under the constructors:

$$\frac{\mathsf{D}_i(\mathbf{t}_1) \quad \cdots \quad \mathsf{D}_i(\mathbf{t}_r)}{\mathsf{D}_i(\mathbf{c}(\mathbf{t}_1 \ldots \mathbf{t}_r))} \qquad \text{for each constructor } \mathbf{c}\text{, of arity } r \geqslant 0\text{, and every } i \geqslant 0$$

Thus the intended semantics is that each D_i is a copy of $\mathbb{A}(C)$.

The data elimination schema has two cases. The main form, *ramified induction,* is

$$\frac{\mathsf{D}_i(\mathbf{t}) \quad \text{CLOSED}_C[\boldsymbol{\varphi}]}{\boldsymbol{\varphi}[\mathbf{t}]}$$

for all formulas $\boldsymbol{\varphi}$ in which no D_j appear with $j \geqslant i$. Here $\text{CLOSED}_C[\boldsymbol{\varphi}]$ states that $\lambda x.\, \boldsymbol{\varphi}[x]$ is closed under the constructors; that is, the conjunction, over the constructors $\mathbf{c}$ of C, of the formulas

$$\forall x_1 \ldots x_r.\ \bigwedge_i \boldsymbol{\varphi}[x_i] \to \boldsymbol{\varphi}[\mathbf{c}(\vec{x})] \qquad r = \underline{arity}(\mathbf{c})$$

A degenerated form of induction is *Reasoning-by-Cases,* which for $\mathbb{N}$ reads

$$\frac{\mathsf{N}(\mathbf{t}) \quad \boldsymbol{\varphi}[\mathbf{0}] \quad \boldsymbol{\varphi}[\mathsf{s}x]}{\boldsymbol{\varphi}[\mathbf{t}]}$$

and for an arbitrary single-sorted algebra $\mathbb{A}(\mathbf{c}_1 \ldots \mathbf{c}_k)$ reads

$$\frac{\mathsf{D}(\mathbf{t}) \quad \boldsymbol{\varphi}[\mathbf{c}_1(x_1 \ldots x_{r_1})] \quad \cdots \quad \boldsymbol{\varphi}[\mathbf{c}_k(x_1 \ldots x_{r_k})]}{\boldsymbol{\varphi}[\mathbf{t}]}$$

For Reasoning-by-Cases our formalism has no ramification conditions, that is

$$\frac{\mathsf{D}_i(\mathbf{t}) \quad \boldsymbol{\varphi}[\mathbf{c}_1(x_1 \ldots x_{r_1})] \quad \cdots \quad \boldsymbol{\varphi}[\mathbf{c}_k(x_1 \ldots x_{r_k})]}{\boldsymbol{\varphi}[\mathbf{t}]}$$

regardless of the tiers occurring in $\boldsymbol{\varphi}$.

A philosophical rationale for ramified formalisms was outlined in [23].

It is also natural to consider *separation axioms*, which guarantee that the denotation of all ground terms are distinct; for $\mathbb{N}$ these are Peano's third and fourth axioms, $\forall x\, \mathsf{s}x \neq \mathbf{0}$ and $\forall x, y\, \mathsf{s}x = \mathsf{s}y \to x = y$. However, these axioms have no effect on the provability of programs, as defined below; see [27].

3.2 Provable functions

An equational program P over $\mathbb{A}(C)$, with principal function-identifier $\mathbf{f}$ of arity r, is *provable* in $\mathbf{RIT}(C)$ if for some $i_1 \ldots i_r$, j the formula

$$\mathbf{D}_{i_1}(x_1) \wedge \cdots \mathbf{D}_{i_r}(x_r) \rightarrow \mathbf{D}_j(\mathbf{f}(\vec{x}))$$

is provable in $\mathbf{RIT}(C)$ from $\forall P$.[5] A function over $\mathbb{A}(C)$ is said to be *provable* in $\mathbf{RIT}(C)$ if it is computed by some program provable in $\mathbf{RIT}(C)$.

It is easy to see that $\mathbf{D}_i(x) \rightarrow \mathbf{D}_j(x)$ is provable whenever $i \geqslant j$. From this it can be shown that the collection of provable functions is closed under composition.

Our main results about the provable functions of ramified intrinsic theories are as follows. Let $\mathbb{A}(C)$ be a non-trivial[6] inductively generated algebra.

THEOREM 3 *A function over $\mathbb{A}(C)$ is provable in $\mathbf{RIT}(C)$ iff it is computable in elementary time, i.e. in time of order $2^{\cdot^{\cdot^{2^n}}}$ for a fixed stack of 2's.*

Let $\mathbf{RIT}^+(C)$ be the sub-formalism of $\mathbf{RIT}(C)$ in which induction is restricted to data-predicative formulas (as defined in §3.2 above).

THEOREM 4 *A function over $\mathbb{A}(C)$ is provable in $\mathbf{RIT}^+(C)$ iff it is computable in polynomial time on a register machine over $\mathbb{A}(C)$, i.e. iff it is poly-time where $\mathbb{A}(C)$ is a word algebra, and linear-space where $\mathbb{A}(C)$ is a unary algebra.*

In Theorems 4 all functions are in fact provable using $\mathbf{D}_0$ and $\mathbf{D}_1$ only, albeit with more complex proofs.

3.3 Provability of functions

We outline first a proof of the backward implications of Theorems 3 and 4, showing that functions in the given complexity classes are provable in the corresponding intrinsic theories.

Refer first to $\mathbb{N}$ and to the standard definitions by recurrence of $+$ and $\times$. It is easy to prove by induction, in $\mathbf{RIT}^+(\mathbb{N})$, that $\mathbf{N}_{i+1}(x) \wedge \mathbf{N}_i(y) \rightarrow \mathbf{N}_i(x+y)$, from which $\mathbf{N}_{i+1}(x) \wedge \mathrm{N}_{i+1}(y) \rightarrow \mathbf{N}_i(x \times y)$. Moreover, our proof above for the function $\mathbf{exp}$ is in $\mathbf{RIT}(\mathbb{N})$, where we rewrite $\mathbf{N}(x)$ as $\mathbf{N}_1(x)$, and rewrite $\mathbf{N}(\cdots)$ as $\mathbf{N}_0(\cdots)$ elsewhere. Since the provable functions (of any of the formalisms considered) are closed under composition, these results show that all polynomials are provable in $\mathbf{RIT}^+(C)$ provided $\mathbb{A}(C)$ is non-trivial,[7] and all elementary functions are provable in $\mathbf{RIT}(C)$, provided $\mathbb{A}(C)$ is not trivial.

[5]It can be shown that other natural definitions are equivalent to this. For instance, one may require $i_1 = \cdots = i_r$ and/or $i_\ell \geqslant j$ without changing the collection of provable functions.

[6]Recall that $\mathbb{A}(C)$ is non-trivial iff it is infinite.

[7]Considerable more work is needed to show that all polynomial are provable using only two data levels; see [22] for an analogous proof for function definition by ramified recurrence.

In our generic setting, the natural machine model is also generic, namely register machines over $\mathbb{A}(C)$, as defined e.g. in [22]. There it is also shown that the transition functions of configurations over register machine are definable by simultaneous ramified recurrence in base type. It is easy to see that such definitions preserve provability in $\mathbf{RIT}^+(C)$. Combined with the availability of elementary-time clocks in $\mathbf{RIT}(C)$, it follows that all function computable in elementary time are provable in $\mathbf{RIT}(C)$. Since polynomial time clocks are available in $\mathbf{RIT}^+(C)$, and Turing machines are trivially simulated by register machines over word algebras, it also follows that all poly-time functions are provable in $\mathbf{RIT}^+(C)$ for any word algebra $\mathbb{A}(C)$. Finally, all linear-space functions over $\mathbb{N}$ (and other unary algebras) are computable in poly-time by a register machine over $\mathbb{N}$ (see [22], the proof idea is due to Gurevich), from which the backward direction of Theorem 4 follows.

3.4 Complexity of provable functions

We now tackle the converse implications in Theorems 3 and 4, showing that provable functions are in the corresponding complexity classes. For proofs in constructive (i.e. intuitionistic) logic, we can use the collapsing Curry-Howard morphism κ of [26], as follows.

Suppose P is a program with principal function-identifier $\mathbf{f}$, computing a function f over $\mathbb{A}(C)$. Let $\mathcal{D}$ be a derivation of $\mathsf{D}_i(\vec{x}) \rightarrow \mathsf{D}_j(\mathbf{f}(\vec{x}))$ from P. First, we may assume without loss of generality that $\mathcal{D}$ is in fact in minimal logic, as shown in [26]. The homomorphism κ then maps $\mathcal{D}$ to a term $\kappa\mathcal{D}$ that defines f. If $\mathcal{D}$ is in $\mathbf{RIT}(C)$ then $\kappa\mathcal{D}$ is a term in an applied λ-calculus with the constructors of C as constants, and with ramified recurrence operators for all ramified types, as defined in [25]. It is shown there that all such functions are computable in elementary time.

If $\mathcal{D}$ is in $\mathbf{RIT}^+(C)$, then $\kappa\mathcal{D}$ is as above, but with ramified recurrence operators of types that are products of the base type.[8] Such λ-recurrence-terms are simply function definitions by ramified recurrence, defined in [22], and shown there to be computable in poly-time by register machines over $\mathbb{A}(C)$. For word algebras these are precisely the poly-time functions, and for unary algebras the linear-space functions.

For provability over classical logic, we are uncertain as of this writing whether the method above can be adapted. While it is easy to map a classical convergence proof to an intuitionistic one (as shown in [26]), the restriction of induction to data-predicative formulas may be violated in the process. An alternative method which does yield the result, albeit at considerably greater effort, is a direct analysis of the complexity of normalization of natural deduction derivations, as developed in [27] for an analogous (un-ramified) result.

[8]In sorted data-systems there may be several base types.

4 Solitary intrinsic theories

4.1 Origins and motivation of solitary deductions

It has been known for long that allowing resources to be invoked only once is related to poly-time computation. For instance, linear logic leads to poly-time [8, 7], second-order existential database queries are poly-time if the matrix is Horn [18, 10], monotone inductive definitions (where each object is inserted only once) define exactly the poly-time queries over finite structures, ramified recurrence with parameters does not lead out of poly-time if only one parameter is used [1], Turing machines operating in poly-space accepts exactly the poly-time languages if non-blank tape-cells cannot be reused, etc.

Continuing in this vein, Martin Hofmann [12] developed a linear-types ramified functional calculus that defines exactly the poly-time non-size-increasing functions, even if recurrence is used at all finite types. Independently, and building on ideas of Jones [14], we showed [24] that allowing abstracted higher order functions to be used only once in λ-recurrence terms, yield exactly poly-time. Here we demonstrate an analogous result for program provability: the provable functions of $\mathbf{IT}(C)$ are exactly the poly-time functions if we allow closing of an assumption (in a natural deduction system) only if it is used once.

Proof theoretic characterizations of poly-time that build on linearity have already appeared, among others in [3]. The main advantage of our present result is that it does not require an overlay of syntactic machinery on the formulas; the proofs we consider are all proofs in intrinsic theories (whose syntax is extremely simple), and the structural properties that they satisfy can be automatically checked.[9] This yields a transparent machinery for certifying program feasibility.

Since poly-time has rather simple proof-theoretic characterizations (e.g. [27] and Theorem 4 above), the main motivation of restricted-multiplicity conditions is the attempt to permit induction for all formulas, thereby providing the user of the formalism (human or automated) with a larger arsenal of methods. Consequently, it is self-defeating to abandon in the process other methods, notably if the latter are important and natural. For instance, taken in isolation, restricted multiplicity disallows a direct and simple proof of the squaring function! Indeed, one would wish to combine the advantages of various approaches, rather than piling up the hurdles of using them. We show below that one can use different restrictions on formulas, depending on whether they correspond under Curry-Howard to first-order or higher-order types. If we restrict the former as in [27], and the latter by the multiplicity condition above, then we add a proof method to our arsenal without losing any.

4.2 A sequential formalization of solitary derivations

Rather than imposing explicit restriction on closing of assumptions in derivations, we can formalize solitary derivability using a substructural calculus. Consider the following natural

[9]Note that we do *not* make this claim for ramified intrinsic theories.

deduction inference rules for the logical operators, exhibited in a sequential style. A *sequent* here is a pair , $\Rightarrow \varphi$ where , is a multi-set of formulas, and φ is a formula. We write φ for $\{\varphi\}$, and , $,\Delta$ for , $\cup\,\Delta$. As usual for multi-sets, the multiplicity in , $\cup\,\Delta$ of a formula ψ is the sum of its multiplicities in , and in Δ.

$$\varphi \Rightarrow \varphi$$

$$\frac{,\ \Rightarrow \varphi \quad \Delta \Rightarrow \psi}{,\ ,\Delta \Rightarrow \varphi \wedge \psi}\ \wedge\mathrm{I} \qquad \frac{,\ \Rightarrow \varphi_0 \wedge \varphi_1}{,\ \Rightarrow \varphi_i}\ \wedge\mathrm{E}_i$$

$$\frac{,\ ,\varphi \Rightarrow \psi}{,\ \Rightarrow \varphi \rightarrow \psi}\ \rightarrow\mathrm{I} \qquad \frac{,\ \Rightarrow \varphi \rightarrow \psi \quad \Delta \Rightarrow \varphi}{,\ ,\Delta \Rightarrow \psi}\ \rightarrow\mathrm{E}$$

$$\frac{,\ \Rightarrow \varphi[z]}{,\ \Rightarrow \forall x\, \varphi[x]}\ \forall\mathrm{I} \qquad \frac{,\ \Rightarrow \forall x\ \ \varphi[x]}{,\ \Rightarrow \varphi[\mathbf{t}]}\ \forall\mathrm{E}$$

$$\frac{,\ \Rightarrow \varphi[\mathbf{t}]}{,\ \Rightarrow \exists x\, \varphi[x]}\ \exists\mathrm{I} \qquad \frac{,\ \Rightarrow \exists x\ \ \varphi[x] \quad \Delta, \varphi[z] \Rightarrow \varphi}{,\ ,\Delta \Rightarrow \varphi}\ \exists\mathrm{E}$$

Note that the rule above for conjunction introduction is the linear logic rule for multiplicative conjunction, whereas the rule of conjunction elimination is the rule for additive conjunction. It seems that strengthening the latter to the multiplicative rule for conjunction elimination,

$$\frac{,\ \Rightarrow \psi_0 \wedge \psi_1 \quad \Delta, \psi_0, \psi_1 \Rightarrow \varphi}{,\ ,\Delta \Rightarrow \varphi}$$

makes no difference as to the set of provable programs, and we could have adopted it instead. However, our approach here is not based on distilling a fragment of linear logic. Indeed, we do not re-interpret the logical connectives as conveying limited use of resources, but limit instead the use of assumptions as logical resources, at the level of the derivation rather than that of the formulas.

4.3 Solitary deductions and poly-time

THEOREM **5** *A function f over $\mathbb{A}(C)$ is provable by a solitary and induction-predicative derivation of* $\mathbf{IT}(C)$ *iff f is computed in poly-time on a register machine over $\mathbb{A}(C)$, i.e. iff f is poly-time where $\mathbb{A}(C)$ is a word algebra, or f is linear-space where $\mathbb{A}(C)$ is a unary algebra.*

Proof Outline. To see that every poly-time function is provable, first observe that the derivation above for multiplication is solitary and predicative. Note that we insist here on using a formulation of function provability as $\forall P$, $\mathbf{D}(\vec{x}) \vdash \mathbf{D}(\mathbf{f}(\vec{x}))$, which for solitary derivations is a more liberal condition than $\mathbf{D}(\vec{x}) \vdash \mathbf{D}(\mathbf{f}(\vec{x}))$: in the former the open assumptions

$\mathbf{D}(\vec{x})$ and $\forall P$ may be used any number of times. This allows iteration of multiplication, whence the provability of polynomial "clock" functions. The configuration-transition functions for register machines, defined in [22], are trivially provable by solitary predicative derivations, completing the backward direction of the proof.

To prove the converse, assume that f is a function over $\mathbb{A}(C)$, provable by a solitary predicative derivation of $\mathbf{IT}(C)$. The proof of [26, Corollary 11] transforms a solitary predicative derivation of $\mathbf{IT}(C)$ based on classical logic to a solitary derivation of $\mathbf{RIT}(C)$ based on minimal logic. The latter is mapped, under the homomorphism κ defined in [27, §3], to a solitary input-driven term in the ramified λ-recurrence-calculus of $\mathbb{A}(C)$, defined in [24], which defines f. As proved there, all functions definable in that calculus are poly-time. $\dashv$

THEOREM **6** *A function f over $\mathbb{A}(C)$ is provable by a solitary derivation of $\mathbf{RIT}(C)$ iff f is computed in poly-time on a register machine over $\mathbb{A}(C)$, i.e. iff f is poly-time where $\mathbb{A}(C)$ is a word algebra, or f is linear-space where $\mathbb{A}(C)$ is a unary algebra.*

Proof Outline. The backward direction is similar to the analogous direction for Theorem 5 above. It is essential here that we have available the un-ramified form of Reasoning-by-Cases, in showing that the configuration-transition functions are provable as tier-preserving (i.e. $\mathbf{D}_0(x) \to \mathbf{D}_0(\mathbf{f}(x))$, so that that the iterate is provable, by Induction.

To prove the converse, assume that f is a function over $\mathbb{A}(C)$, provable by a solitary derivation of $\mathbf{RIT}(C)$. We proceed as in the proof of Theorem 5 above, to obtain a solitary term in the ramified version of the λ-recurrence calculus for $\mathbb{A}(C)$. Here we need an extension of the result of [24], where recurrence need not be input-driven, but instead is ramified. A straightforward induction on the tier of the Ramified Induction concludes the proof. (The proof in [24] gives the induction step of the meta-proof.) $\dashv$

4.4 Combining ramification and non-multiplicity

In Theorems 5 and 6 we showed that we can trade the restriction of induction to positive formulas for a prohibition of assumption multiplicity. The class of provable functions is poly-time in either case. While this result is potentially beneficial in some cases, it seems that multiple invocation of assumptions is, in fact, used and needed in actual proofs far more frequently than induction over non-positive formulas.[10] Fortunately, the two restriction are orthogonal, and we can relax the multiplicity restriction to non-data-negative formulas.

THEOREM **7** *Let $\mathbf{T}$ be either predicative $\mathbf{IT}(C)$ or $\mathbf{RIT}(C)$. Let $\mathbf{T}'$ be $\mathbf{T}$, where multiple assumptions are closed only for data-non-negative formulas. Then the provable functions of $\mathbf{T}$ are precisely the functions computable in poly-time on a register machine over $\mathbb{A}(C)$.*

[10]We may trivially permit in $\mathbf{RIT}^+(C)$ induction also on formulas with only negative occurrences of $\mathbf{D}$, which include the renditions of $\mathbf{\Pi}^0_1$ formulas.

The proof that all provable functions are poly-time follows the pattern in the proofs above, with a significant elaboration of the λ-recurrence calculi considered. We omit the detail, and just point out the essential feature of the situation: when closing multiple occurrence of an assumption is prohibited for non-data-negative formulas, inductions must either be for non-data-negative formulas, or else have solitary proofs for the premises.

4.5 Extensions and future research

Our work in progress focuses on two directions in the use of intrinsic theories. Of particular interest is the development of intrinsic theories for analysis, i.e. the second order theories of inductive data systems in general, and word algebras and $\mathbb{N}$ in particular. In [21] we showed that second order logic with comprehension (i.e. set existence) restricted to relations definable by positive existential formulas, yields poly-time in an appropriate sense. We believe that second order variants of intrinsic theories for poly-time, with similarly restricted existence principles for functions and relations, also yield only poly-time provable functions. Such developments would have several benefits. At the meta-logic level, such theories provide a framework for defining and studying feasibility of higher-type functionals, extensively studied recently (see e.g. [15, 13]). Also, there is broad potential for formalizing mathematical analysis in such theories, with a twofold benefit: the computational feasibility of certain constructions would fall out automatically as a result, and, more importantly yet, the methodology would automatically extract feasible programs from proofs in intrinsic theories.

Another thread in progress is the development of intrinsic theories that correspond to poly-space. The method is to slightly relax the condition of solitary derivations, and to permit assumption multiplicity at least across the cases of degenerate induction (deduction by cases), and perhaps across all minor premises of induction in general as well as conjunction. Interestingly, such substructural formalisms have no simple correspondence to linear logic, albeit they are based on "non-reused" resources.

References

[1] A. Beckmann and A. Weiermann. A term rewriting characterization of the polytime functions and related complexity classes. *Archive for Mathematical Logic*, 36:11–30, 1996.

[2] Stephen Bellantoni and Stephen Cook. A new recursion-theoretic characterization of the poly-time functions. *Computational Complexity*, 1992.

[3] Stephen Bellantoni and Martin Hofmann. A new feasible arithmetic. *Journal for Symbolic Logic*, 2001.

[4] Stephen J. Bellantoni. Ranking arithmetic proofs by implicit ramification. In P. Beame and S. Buss, editors, *Proof complexity and feasible arithmetic*, DIMACS Series in Discrete Mathematics v. 39, 1998.

[5] Stephen J. Bellantoni, Karl-Heinz Niggl, and Helmut Schwichtenberg. Higher type recursion, ramification and polynomial time. *Annals of Pure and Applied Logic*, 104 (1-3):17–30, 2000.

[6] Samuel Buss. *Bounded Arithmetic.* Bibliopolis, Naples, 1986.

[7] Jean-Yves Girard. Light linear logic. *Information and Computation*, 143, 1998.

[8] Jean-Yves Girard, Andre Scedrov, and Philip Scott. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science*, 97:1–66, 1992.

[9] Kurt Gödel. Über eine bisher noch nicht benutzte erweiterung des finiten standpunktes. *Dialectica*, 12:280–287, 1958.

[10] E. Grädel. Capturing Complexity Classes by Fragments of Second Order Logic. *Theoretical Computer Science*, 101:35–57, 1992.

[11] Martin Hofmann. A mixed modal/linear lambda calculus with applications to bellantoni-cook safe recursion. In *Proceedings of CSL'97*, pages 275–294. Springer-Verlag LNCS 1414, 1998.

[12] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. In *Proceedings of LICS'99*, pages 464–473. IEEE Computer Society, 1999.

[13] R. Irwin, B.M. Kapron, and J. Royer. On characterizations of the basic feasible functionals, part i. *Journal of Functional Programming*, 2001.

[14] N. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, Cambridge, MA, 1997.

[15] Bruce Kapron and Stephen A. Cook. A new characterization of type-2 feasibility. *SIAM J. on Computing*, 25(1):117–132, 1996.

[16] Jan Krajicek. *Bounded Arithmetic, Propositional Logic, and Complexity Theory.* Cambridge University Press, Cambridge, 1995.

[17] D. Leivant. Predicative recurrence in finite type. In A. Nerode and Yu.V. Matiyasevich, editors, *Logical Foundations of Computer Science (Third International Symposium)*, LNCS, pages 227–239, Berlin, 1994. Springer-Verlag.

[18] Daniel Leivant. Descriptive characterizations of computational complexity. In *Second Annual Conference on Structure in Complexity Theory*, pages 203–217, Washington, 1987. IEEE Computer Society Press. Revised in *Journal of Computer and System Sciences*, 39:51–83, 1989.

[19] Daniel Leivant. Contracting proofs to programs. In P. Odifreddi, editor, *Logic and Computer Science*, pages 279–327. Academic Press, London, 1990.

[20] Daniel Leivant. Subrecursion and lambda representation over free algebras. In Samuel Buss and Philip Scott, editors, *Feasible Mathematics*, Perspectives in Computer Science, pages 281–291. Birkhauser-Boston, New York, 1990.

[21] Daniel Leivant. A foundational delineation of poly-time. *Information and Computation*, 110:391–420, 1994. (Special issue of selected papers from LICS'91, edited by G. Kahn). Preminary report: A foundational delineation of computational feasibility, in Proceedings of the Sixth IEEE Conference on Logic in Computer Science, IEEE Computer Society Press, 1991.

[22] Daniel Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In Peter Clote and Jeffrey Remmel, editors, *Feasible Mathematics II*, Perspectives in Computer Science, pages 320–343. Birkhauser-Boston, New York, 1994. Postscript source: www.cs.indiana.edu/hyplan/leivant/papers/ramified1.ps.

[23] Daniel Leivant. Intrinsic theories and computational complexity. In D. Leivant, editor, *Logic and Computational Complexity*, LNCS, pages 177–194, Berlin, 1995. Springer-Verlag.

[24] Daniel Leivant. Applicative control and computational complexity. In J. Flum and M. Rodriguez-Artalejo, editors, *Computer Science Logic (Proceedings of the Thirteenth CSL Conference*, pages 82–95, Berlin, 1999. Springer Verlag (LNCS 1683). Postscript source: www.cs.indiana.edu/hyplan/leivant/papers/lambda-control.ps.

[25] Daniel Leivant. Ramified recurrence and computational complexity III: Higher type recurrence and elementary complexity. *Annals of Pure and Applied Logic*, 96/3:209–229, 1999. Special issue in honor of Rohit Parikh's 60th Birthday; editors: M. Fitting, R. Ramanujam and K. Georgatos).

[26] Daniel Leivant. Intrinsic reasoning about functional programs I: first order theories. *Annals of Pure and Applied Logic*, 2001. Postscript source: www.cs.indiana.edu/hyplan/leivant/papers/intrinsic-fo.ps.

[27] Daniel Leivant. Termination proofs and complexity certification. Postscript source: www.cs.indiana.edu/hyplan/leivant/papers/intrinsic-pr.ps, February 2001.

[28] Harold Simmons. The realm of primitive recursion. *Archive for Mathematical Logic*, 27:177–188, 1988.

System Presentation: An Analyser of rewriting systems complexity

J.-Y. Moyen [*]

May 10, 2001

Abstract

This paper briefly describes ICAR, a program which analyses the implicit complexity of first order functionnal programs. ICAR is based on two previous characterisation of PTIME and PSPACE by mean of term rewriting termination orderings and polynomial quasi-interpretations.

1 The analyser ICAR

We shall consider term rewriting systems build over three distinct sets: function symbols (defined symbols), constructors and variables. The function symbols are ordered by a precedence $\prec_{\mathcal{F}}$ and constructors are considered as the smaller elements of the precedence. A program is a set of rewriting rules.

ICAR (Implicit Complexity AnalyseR) first checks the termination of the given rewriting system and then tries to find a bound on its complexity. This work is based on [3, 4, 1] for complexity analysis. Program transformation by the mean of memoization is based on Jones work [2].

- **Termination** is check using termination ordering, either the Multiset Path Ordering or the Lexicographic Path Ordering.
- **Complexity** may then be determined by combining the termination ordering used and quasi-interpretations. ICAR may be able to tell that the computed function is in PTIME or PSPACE.
- One of the main interest of our approach is that this analysis gives an upper bound on the complexity of the function computed rather than on the complexity of the program. This kind of complexity analysis was dubbed *implicit*. So ICAR also gives a way (*i.e.* a new operationnal semantics) to effectively achieve this bound.

One may then run the program using different operationnal semantics and verify experimentally the theoretical bound previously obtained.

[*]Loria, Calligramme project, B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France, `moyen@loria.fr`

$$\frac{s \preceq_{mpo} t_i}{s \prec_{mpo} f(\ldots, t_i, \ldots)}\, f \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{s_i \prec_{mpo} \mathtt{g}(t_1, \cdots, t_n) \quad f \prec_{\mathcal{F}} \mathtt{g}}{f(s_1, \cdots, s_m) \prec_{mpo} \mathtt{g}(t_1, \cdots, t_n)}\, \mathtt{g} \in \mathcal{F}, f \in \mathcal{F} \bigcup \mathcal{C}$$

$$\frac{\{s_1, \cdots, s_n\} \prec^m_{mpo} \{t_1, \cdots, t_n\} \quad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g}}{\mathtt{g}(s_1, \cdots, s_n) \prec_{mpo} \mathtt{f}(t_1, \cdots, t_n)}\, \mathtt{f}, \mathtt{g} \in \mathcal{F}$$

Figure 1: Multiset Path Ordering

2 Termination ordering and Quasi-interpretations

2.1 Termination orderings

Definition 1. Let $\prec$ be an ordering over terms. The *multiset extension* $\prec^m$ of $\prec$ is defined as follow:
$M = \{m_1, \cdots, m_k\} \prec^m \{n_1, \cdots, n_k\} = N$ if and only if $M \neq N$ and there exists a permutationπ such that:

- There exists j such that $m_j \prec n_{\pi(j)}$
- For all i, $m_i \preceq n_{\pi(i)}$

Definition 2. The *Multiset Path Ordering* (MPO) is defined in the rules of Figure 1.

A MPO-program is a program such that for each rule $l \to r$, $r \prec_{mpo} l$.

Definition 3. Let $\prec$ be an ordering over terms. The *lexicographic extension* $\prec^l$ of $\prec$ is recursively defined as follow:

$(t_1, \cdots, t_n) \prec^l (s_1, \cdots, s_m)$ if and only if $t_1 \prec s_1$ or $t_1 = s_1$ and $(t_2, \ldots, t_n) \prec^l (s_2, \ldots, s_m)$.

Definition 4. The *Lexicographic Path Ordering* (LPO) is defined in the rules of Figure 2.

A LPO-program is a program such that for each rule $l \to r$, $r \prec_{lpo} l$.

$$\frac{s \preceq_{lpo} t_i}{s \prec_{lpo} \mathtt{f}(\ldots, t_i, \ldots)}$$

$$\frac{s_i \prec_{lpo} \mathtt{f}(t_1, \cdots, t_n) \quad \mathtt{g} \prec_{\mathcal{F}} \mathtt{f}}{\mathtt{g}(s_1, \cdots, s_m) \prec_{lpo} \mathtt{f}(t_1, \cdots, t_n)}$$

$$\frac{(s_1, \cdots, s_n) \prec_{lpo}^{l} (t_1, \cdots, t_n) \quad \mathtt{f} \approx_{\mathcal{F}} \mathtt{g} \quad s_j \prec_{lpo} \mathtt{f}(t_1, \cdots, t_n)}{\mathtt{g}(s_1, \cdots, s_n) \prec_{lpo} \mathtt{f}(t_1, \cdots, t_n)}$$

Figure 2: Lexicographic Path Ordering

2.2 Quasi-interpretation

Definition 5. A *quasi-interpretation* of a symbol f is a function $(\!|f|\!)$ such that:

- $(\!|f|\!)$ is bounded by a polynomial
- $(\!|f|\!)$ is (non strictly) increasing
- $(\!|f|\!)(X_1, \cdots, X_n) \geq X_i$ for all $i \leq n$.
- $(\!|\mathbf{c}|\!)(X_1, \cdots, X_n) = \sum_{i=1}^{n} X_i + \gamma$ for all constructors $\mathbf{c}$ where $\gamma > 0$ is a constant.

Quasi-interpretations are extended to terms as usual:

$$(\!|f(t_1, \cdots, t_n)|\!) = (\!|f|\!)((\!|t_1|\!), \ldots, (\!|t_n|\!))$$

A program admits a quasi-interpretation if for each rule $l \to r$, $(\!|r|\!) \leq (\!|l|\!)$. This is clearly not sufficient for termination.

2.3 Theorems

Theorem 6. *The set of functions computable by MPO-programs admiting quasi-interpretations is* exactly PTIME*, the set of functions computable in polynomial time.*

Proof. See [4]. □

Theorem 7. *The set of functions computable by LPO-programs admiting quasi-interpretations is* exactly PSPACE*, the set of functions computable in polynomial space.*

Proof. See [1] □

3 Implementation

There are two things to implement. First, determining whether the program terminates by MPO or LPO and second determining if it admits a quasi-interpretation.

The termination using the orderings is a well-known problem. It is known to be PTIME computable if the precedence is given and NP-complete if the precedence has to be found. Fortunatly ICAR uses a restriction of the usual orderings: a constructor always has a precedence smaller than any function symbol.

Claim 8. *With this restriction, the precedence can be found in polynomial time. So termination by either MPO or LPO can be checked in time* $2^{c \times k}$ *where* k *is the maximal arity of a function symbol and* c *is a constant.*

Proof. See section 4 □

The main difficulty lies in the second part. Indeed, one doesn't know if quasi-interpretations are decidable. The similar problem with strict inegalities (*i.e.* finding polynomial interpretations) seems to be undecidable. Fortunatly, quasi-interpretations are quite easy to find because an upper bound on the program denotation turns to be a good candidate. So, even if finding one is a hard task for a computer, its an easy job for the programmer. The idea is to provide a potential quasi-interpretation together with the rewriting rules, and the program just has to check it.

Of course, there exists programs able to deal with symbolic computation. So the obvious way to check quasi-interpretations is to deleguate this job to such a program. Currently, ICAR uses Maple as a quasi-interpretation checker. Maple may be unable to check some inequality (especially those using a lot of *max*s. As far as I know, there isn't any software able to treat them properly, but one is under devellopement by Fabrice Rouiller at LIP6 (Paris) and should be available by the end of the year.

Until this moment, ICAR returns the quasi-interpretations that Maple was unable to solve and hopes that the user will be less clumsy.

Example 9.

1. the following program computes the addition and the multiplication of two unary numbers.

$$\begin{aligned}
\texttt{add}(\mathbf{0}, y) &\to y \\
\texttt{add}(\mathbf{S}(x), y) &\to \mathbf{S}(\texttt{add}(x, y)) \\
\texttt{mult}(\mathbf{0}, y) &\to \mathbf{0} \\
\texttt{mult}(\mathbf{S}(x), y) &\to \texttt{add}(y, \texttt{mult}(x, y))
\end{aligned}$$

It terminates either by MPO or LPO by putting `add` $\prec_{\mathcal{F}}$ `mult` and admits a quasi-interpretation: $(\!|\mathtt{add}|\!)(X,Y) = X+Y$, $(\!|\mathtt{mult}|\!)(X,Y) = X \times Y$, $(\!|\mathbf{0}|\!) = 1$, $(\!|\mathbf{S}|\!)(X) = X+1$. Let's verify the quasi-interpretaion for the last rule:

$$\begin{aligned}(\!|\mathtt{mult}(\mathbf{S}(x),y)|\!) &= (\!|\mathtt{mult}|\!)((\!|\mathbf{S}(x)|\!),Y) = (X+1)\times Y\\ (\!|\mathtt{add}(y,\mathtt{mult}(x,y))|\!) &= Y + (\!|\mathtt{mult}(x,y)|\!) = Y + X \times Y\end{aligned}$$

2. one may computes the length of the longest common subsequence of two string as follow:

$$\begin{aligned}\mathtt{max}(n,\mathbf{0}) &\to n\\ \mathtt{max}(\mathbf{0},m) &\to m\\ \mathtt{max}(\mathbf{S}(n),\mathbf{S}(m)) &\to \mathbf{S}(\mathtt{max}(n,m))\end{aligned}$$

$$\begin{aligned}\mathtt{lcs}(x,\boldsymbol{\epsilon}) &\to \mathbf{0}\\ \mathtt{lcs}(\boldsymbol{\epsilon},y) &\to \mathbf{0}\\ \mathtt{lcs}(\mathbf{a}(x),\mathbf{a}(y)) &\to \mathbf{S}(\mathtt{lcs}(x,y))\\ \mathtt{lcs}(\mathbf{b}(x),\mathbf{b}(y)) &\to \mathbf{S}(\mathtt{lcs}(x,y))\\ \mathtt{lcs}(\mathbf{a}(x),\mathbf{b}(y)) &\to \mathtt{max}(\mathtt{lcs}(x,\mathbf{b}(y)),\mathtt{lcs}(\mathbf{a}(x),y))\\ \mathtt{lcs}(\mathbf{b}(x),\mathbf{a}(y)) &\to \mathtt{max}(\mathtt{lcs}(x,\mathbf{a}(y)),\mathtt{lcs}(\mathbf{b}(x),y))\end{aligned}$$

By putting `max` $\prec_{\mathcal{F}}$ `lcs`, this is a MPO program. It admits a quasi-interpretation: $(\!|\mathbf{0}|\!) = (\!|\boldsymbol{\epsilon}|\!) = 1$, $(\!|\mathbf{S}|\!)(X) = (\!|\mathbf{i}|\!)(X) = (\!|\mathbf{j}|\!)(X) = X+1$, $(\!|\mathtt{max}|\!)(X,Y) = (\!|\mathtt{lcs}|\!)(X,Y) = \max(X,Y)$.

So the program computes a function in PTIME. Note that the explicit complexity of the program is exponential. The polynomial bound is obtained by the mean of memoization: the operational semantics is modified as shown in Figure 3. Every time a function call is computed, its result is stored in a cache and will be reused directly if the same call is needed another time.

ICAR is able to compute the value of a term with or without using the cache. It keeps a trace of the time and space used by the computation (*i.e.* the number of reduction steps and the maximum size of the cache and the environnement). The results for computing $\mathtt{lcs}(\mathbf{a}^n(\boldsymbol{\epsilon}),\mathbf{b}^n(\boldsymbol{\epsilon}))$ are:

n	`Call-by-value`		`Memoization`	
	`time`	`space`	`time`	`space`
0	4	1	4	1
1	17	2	17	5
2	63	2	48	10
3	219	2	97	17
4	771	2	164	26
5	2775	2	249	37
6	10169	2	352	50

$$\frac{\sigma(x) = v}{\mathcal{E}, \sigma \vdash \langle C, x\rangle \to \langle C, v\rangle}$$

$$\frac{\mathbf{c} \in \mathcal{C} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathbf{c}(t_1, \cdots, t_n)\rangle \to \langle C_n, \mathbf{c}(v_1, \cdots, v_n)\rangle}$$

$$\frac{\mathtt{f} \in \mathcal{F} \quad \mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle \quad (\mathtt{f}(v_1, \cdots, v_n), v) \in C_n}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \to \langle C_n, v\rangle}$$

$$\frac{\mathcal{E}, \sigma \vdash \langle C_{i-1}, t_i\rangle \to \langle C_i, v_i\rangle \ \mathtt{f}(\vec{p}) \to r \in \mathcal{E} \ p_i\sigma' = v_i \ \mathcal{E}, \sigma' \vdash \langle C_n, r\rangle \to \langle C, v\rangle}{\mathcal{E}, \sigma \vdash \langle C_0, \mathtt{f}(t_1, \cdots, t_n)\rangle \to \langle C \cup (\mathtt{f}(v_1, \cdots, v_n), v), v\rangle}$$

Figure 3: Call by Value interpreter with cache

4 Finding the precedence

This section describes the implementation of the two main difficulties of ICAR: finding the precedence for the termination orderings and finding the permutation for MPO. Both can be solved in polynomial time thanks to the restriction done upon the precedence.

Lemma 10. *Let $t = f(u_1, \cdots, u_n)$ and $s = g(v_1, \cdots, v_n)$ be two terms. If $g \approx_{\mathcal{F}} f$ implies $s \prec_{mpo} t$ then $g \prec_{\mathcal{F}} f$ implies $s \prec_{mpo} t$.*

Proof. If t and s are ordered when $\mathtt{g} \approx_{\mathcal{F}} \mathtt{f}$ then for all $i, 1 \leq i \leq n$, there exist a $j, 1 \leq j \leq n$ such that $v_i \preceq_{mpo} u_j$ (by definition of MPO). So for all i, $v_i \prec_{mpo} t$, so the two terms are ordered if $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$. □

Lemma 11. *Let $t = f(u_1, \cdots, u_n)$ and $s = g(v_1, \cdots, v_n)$ be two terms. If $g \approx_{\mathcal{F}} f$ implies $s \prec_{lpo} t$ then $g \prec_{\mathcal{F}} f$ implies $s \prec_{lpo} t$.*

Proof. By definition of LPO, the hypothesis implies $v_i \prec_{lpo} t$. □

Lemma 12. *Let $t = f(u_1, \cdots, u_n)$ and $s = g(v_1, \cdots, v_n)$ be two terms. If $s \prec_{mpo} t$ or $s \prec_{lpo} t$, no symbol (function or constructor) in s may have a precedence greater than the greatest symbol in t.*

Proof. Obvious since the symbol with the greatest precedence will lead to the greatest term. □

Lemma 13. *Let $l \to r$ be a rewriting rule such that $r \prec_{mpo} l$ or $r \prec_{lpo} l$. No symbol in r may have a precedence greater than the head symbol of l.*

Proof. As l is the left-hand-side of a rewriting rule, $l = \mathtt{f}(p_1, \cdots, p_n)$ where p_i are patterns, that is terms build only over variables and constructors. Since constructors have a precedence smaller than any function symbol, f has the maximal precedence of l. So by previous lemma, it must also have the maximal precedence of r. □

Corollary 14. *Finding the precedence is performed in polynomial time.*

Proof. By examining rewriting rules and by lemma 13, we obtain a set of constraints of the form $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$. Then, graph-reachability between any two symbol tells weather $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$ or not. If there is both $\mathtt{f} \preceq_{\mathcal{F}} \mathtt{g}$ and $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$, then there must be $\mathtt{f} \approx_{\mathcal{F}} \mathtt{g}$. If there is only $\mathtt{g} \preceq_{\mathcal{F}} \mathtt{f}$, then lemma 10 and 11 tell that one won't looze anything by choosing $\mathtt{g} \prec_{\mathcal{F}} \mathtt{f}$. So the precedence is found in polynomial time. □

References

[1] Bonfante, G., Marion, J.-Y., and Moyen, J.-Y. On lexicographic termination ordering with space bound certifications. Tech. rep., Loria, Mar 2001. accepted to Andre Ershov Fourth International Conference (PSI01).

[2] Jones, N. The expressive power of higher order types or, life without cons. to appear, 2000.

[3] Marion, J.-Y. Analysing the implicit complexity of programs. *Information and Computation* (2000). to appear.

[4] Marion, J.-Y., and Moyen, J.-Y. Efficient first order functional program interpreter with time bound certifications. In *LPAR* (Nov 2000), vol. 1955 of *Lecture Notes in Computer Science*, Springer, pp. 25–42.

On the computational complexity of stack programs

Lars Kristiansen[*] Karl-Heinz Niggl[†]

Abstract

A restricted imperative stack programming language L over an arbitrary but fixed alphabet Σ is considered. The paper presents a purely syntactical method for analysing the impact of nesting loops in L-programs on computational complexity. This gives rise to a measure μ on L-programs, i.e. a function that assigns to each L-program `P` a natural number μ(`P`) computable from the syntax of `P`. It is shown that a function over Σ^* is computable by a Turing machine in polynomial time if and only if it is L-computable with μ-measure 0. More generally, it is shown that a Turing machine runs in time $b(n)$ (where n is the size of the input) for some function b in Grzegorczyk class $\mathcal{E}^{n+2}$ if and only if it can be simulated by an L-program with μ-measure n.

1 Introduction

We study a restricted imperative stack programming language L over an arbitrary but fixed alphabet Σ. Programs in L contain variables `X`, `Y`, `Z`, say, which serve as stacks, each holding an arbitrary word over Σ which can be manipulated by running a program in L. Programs in L are built from primitive instructions `push(a,X)` for `a` $\in \Sigma$, `pop(X)`, `nil(X)` by sequencing $\texttt{P}_1$`;` $\texttt{P}_2$, conditional statements `if top(X)` $\equiv$ `a [Q]` and loop statements `foreach X [Q]`. The operational semantics of L-programs is fairly standard, except possibly that of loop statements. Here we follow a call-by-value semantics that allows one to inspect every symbol on the control stack `X` while preserving its contents.

We are interested in analysing the impact of nesting loops on computational complexity. Obviously, some nesting of loops cause no blow up in computational complexity while others do. So the point in question is: Can one extract information out of the syntax of L-programs so as to separate programs which run in polynomial time (in the size of the input) from programs which do not? And if "Yes", is there a general rationale behind, and how far does it go?

In this paper we propose a purely syntactical method we call μ-measure that assigns to each L-program `P` a natural number μ(`P`) computable from the syntax of `P`. Answering the first question above, we show that the functions over Σ^* computable by a Turing machine in polynomial time are precisely the functions computable by an L program with μ-measure 0. This is an instance of a more general result that answers the second question above: A

[*]Oslo University College, Faculty of Engineering, Norway, e-mail: `larskri@iu.hio.no`.

[†]Technische Universität Ilmenau, Institut für Theoretische und Technische Informatik, PF 100565, 98684 Ilmenau, Germany, email: `niggl@theoinf.tu-ilmenau.de, tel: +49 3677-691444, fax: +49 3677-691237.`

Turing machine M runs in time $b(n)$ for some function b in Grzegorczyk class $\mathcal{E}^{n+2}$ if and only if M can be simulated by an L-program with μ-measure n.

To exemplify the main ideas behind the measure μ, we first need to explain in more detail the operational semantics of the loop concept in L. For loop statements `foreach X [P]` we require that `P` has no occurrence of imperatives `push(a,X)`, `pop(X)` or `nil(X)`. Thus, the control stack `X` can not be altered during an execution of the loop. However, in order to provide access to each symbol on the control stack `X` during an execution of the loop, the operational semantics of `foreach X [P]` is that of the sequence

$$\texttt{U} := \texttt{X};\ \texttt{P};\ \texttt{pop(X)};\ \ldots;\ \texttt{P};\ \texttt{pop(X)};\ \texttt{X} := \texttt{U}$$

with $|w|$ occurrences of `P; pop(X)` whenever w is stored in `X` before the execution of the loop, where `U` is some reserved variable that is not allowed to occur elsewhere.

It is obvious that we have to nest loops to a certain depth in order to obtain programs of a certain high computational complexity. It is also obvious that some programs with "high loop nesting depth" like e.g.

```
P₁ :≡ foreach X [foreach X [foreach X [foreach X [foreach X [push(a,Y)]]]]]
```

($\texttt{a} \in \Sigma$) run in polynomial time. So "high loop nesting depth" is a necessary condition for high computational complexity, but it is not a sufficient condition. In this paper we give syntactical criteria that separate loops which cause a blow up in computational complexity from those which do not. To outline the main ideas, consider the following two programs:

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

Observe that both $\texttt{P}_2$ and $\texttt{P}_3$ have nesting depth 2, and they look quite similar. However, $\texttt{P}_2$ runs in exponential time while $\texttt{P}_3$ runs in polynomial time, for if w is initially stored in `X`, then the word $a^{2^{|w|}}$ is stored in `Z` after $\texttt{P}_2$ is executed, while $a^{|w|\cdot(|w|+1)}$ is stored in `Z` after $\texttt{P}_3$ is executed. The gist of the matter lies in a (control) *circle* contained inside the outermost loop in $\texttt{P}_2$: Inside the loop controlled by `X`, first *`Y` controls `Z`* in that `Z` is updated via `push(a,Z)` inside a loop controlled by `Y`, and then *`Z` controls `Y`* in the same sense. In contrast, there is no such circle in $\texttt{P}_3$. In fact, it will turn out that the Turing machines with polynomial running time correspond exactly to those programs in L which do not contain a loop $\texttt{foreach X [Q}_1;\ldots;\texttt{Q}_l\texttt{]}$ where the body $\texttt{Q}_1;\ldots;\texttt{Q}_l$ ($l \geq 2$) contains a circle. All these programs will receive μ-measure 0.

These ideas generalise uniformly to all levels of computational complexity as given in the Grzegorczyk hierarchy. We just focus on the critical case where `P` is a loop `foreach X [Q]` and assume that we have already determined $\mu(\texttt{Q})$. Suppose that `Q` is a sequence $\texttt{Q}_1;\ldots;\texttt{Q}_l$, in which case $\mu(\texttt{Q})$ is $\max\{\mu(\texttt{Q}_1),\ldots,\mu(\texttt{Q}_l)\}$. Then we obtain a blow up in computational complexity if *`Q` has a top circle*, that is, `Q` has a circle with respect to a control variable `Y` of some component $\texttt{Q}_i$ with maximal μ-measure $\mu(\texttt{Q})$. In this case, we define $\mu(\texttt{P}) := 1 + \mu(\texttt{Q})$.

In all other cases for Q we define $\mu(\mathtt{P}) := \mu(\mathtt{Q})$, for as we show, these loops do not cause a blow up in computational complexity.

Adding that imperatives have μ-measure 0, one easily verifies for the examples above that $\mu(\mathtt{P}_1) = \mu(\mathtt{P}_3) = 0$ while $\mu(\mathtt{P}_2) = 1$.

The measure μ is convenient for various reasons: Firstly, it operates on an imperative stack programming language L which is very close to restricted Turing machine programming, however, supporting a clear control structure. Secondly, it can be easily extended to extensions of L providing features supported by many high level programming languages. Thirdly, the measure μ is conceptually simple and it characterises computationally relevant complexity classes, thus it can help to ground the concepts of computational complexity by providing a reference point other than the original resource-based concepts. Finally, one can argue that the measure μ is likely to give the minimal complexity for a great deal of natural algorithms, and furthermore, it admits significantly more algorithms in each complexity class than any other known complexity measure on loop programs like "counting nesting depth".

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

This paper builds on recent work on *ramified analysis of recursion* by Bellantoni and Niggl [6], and Niggl [18]. There a purely syntactical method for analysing the impact of nesting (unrestricted) recursions on computational complexity has been proposed, in the context of ordinary schemata-based definitions in [6], in the context of lambda terms over ground-type variables in [18]. Ramified analysis of recursion characterises uniformly the Grzegorczyk hierarchy at and above the linear-space level when based on primitive recursion. One obtains the same hierarchy of classes except with the polynomial-time computable functions at the first level when primitive recursion is replaced with recursion on notation.

Various ramification concepts as initialised by Simmons [23], Leivant [11, 12, 13], Bellantoni and Cook [1] have led to resource-free, purely functional characterisations of many complexity classes, such as the polynomial-time computable functions [1, 15, 14], the linear-space computable functions [2, 13, 19], NC^1 and polylog space [5], NP and the poly-time hierarchy [3], the Kalmár-elementary functions [20], and the exponential time functions of linear growth [8], among many others.

Ramification concepts have also proved fruitful in characterising complexity classes by higher type recursion, such as the Kalmár-elementary functions [16], poly-space [17], and recently the polynomial-time computable functions [4, 10].

2 Preliminaries

We assume only basic knowledge about subrecursion theory, in particular with the Grzegorczyk hierarchy. Readers unfamiliar with these subjects are referred to Grzegorczyk [9], Rose [22] and Clote [7]. We summarise some basic definitions and facts from Rose.

For unary functions f, f^k denotes *kth iterate of* f, i.e. $f^0(x) = x$ and $f^{k+1}(x) = f(f^k(x))$. The sequence of *principal functions* $E_1, E_2, E_3, \dots$ defined by $E_1(x) = x^2 + 2$ and $E_{n+2}(x) = E^x_{n+1}(2)$, enjoys the following *monotonicity* properties: $E_{n+1}(x) \geq x + 1$, $E_{n+1}(x + 1) \geq E_{n+1}(x)$, $E_{n+2}(x) \geq E_{n+1}(x)$ 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* $\mathcal{E}^n$, for $n \geq 2$, is defined as the least class containing the initial functions zero, the successor function, the projection functions and E_{n-1}, and closed under composition and bounded recursion.

By Ritchie [21] the class $\mathcal{E}^2$ characterises the class FLINSPACE of functions computable by a Turing machine in linear space. The class $\mathcal{E}^3$ characterises the Kalmár-elementary functions. Every $f \in \mathcal{E}^n$ satisfies $f(\vec{x}) \leq E^m_{n-1}(\max(\vec{x}))$ for a constant m. 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 $\mathcal{E}^{n+1}$. The union of all these classes characterises the primitive recursive functions.

3 The programming language L

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

Definition 3.1 (L-programs). *L-programs* `P` are inductively defined as follows:

- Every *imperative* `push(a,X)`, `pop(X)`, `nil(X)` is an L-program.
- If $\mathtt{P}_1, \mathtt{P}_2$ are L-programs, then so is the *sequence* statement $\mathtt{P}_1$`;` $\mathtt{P}_2$.
- If `P` is an L program, then so is every *conditional* statement `if top(X)`$\equiv$`a [P]`.
- If `P` is an L-program with no occurrence of `push(a,X)`, `pop(X)` or `nil(X)`, then so is the *loop* statement `foreach X [P]`.

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

Note 3.2. *Every L-program can be written uniquely in the form $\mathtt{P}_1; \ldots; \mathtt{P}_k$ such that each component $\mathtt{P}_i$ is either a loop or an imperative, or else a conditional, and where $k = 1$ whenever `P` is an imperative or a loop or a conditional.*

We will use informal Hoare-like sentences to specify or reason about L-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 `P` is executed, then the condition given by the sentence B is fulfilled after the execution of `P`. For example, $\{\vec{\mathtt{X}} = \vec{w}\}\,\mathtt{P}\,\{\vec{\mathtt{X}} = \vec{w}'\}$ reads as *if the words $\vec{w}$ are stored in the stacks $\vec{\mathtt{X}}$, respectively, before the execution of* `P`*, then $\vec{w}'$ are stored in $\vec{\mathtt{X}}$ after the execution of* `P`. 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 words $\vec{w}$ are stored in the stacks $\vec{\mathtt{X}}$, respectively, before the execution of* `P`*, then each word stored in X_i after the execution of* `P` *has a length bounded by $f_i(|\vec{w}|)$.* Here f_i is any function over $\mathbb{N}$, and $|\vec{w}|$ abbreviates as usual the list $|w_1|, \ldots, |w_n|$.

Definition 3.3 (Operational semantics of L-programs). The operational semantics of L-programs is defined inductively as follows, where X is an arbitrary variable, w denotes an arbitrary word over Σ, a an arbitrary letter in Σ, and ε the *empty word.*

- $\{\mathtt{X} = w\}$ push(a,X) $\{\mathtt{X} = w\mathtt{a}\}$.
- $\{\mathtt{X} = w\mathtt{a}\}$ pop(X) $\{\mathtt{X} = w\}$ and $\{\mathtt{X} = \varepsilon\}$ pop(X) $\{\mathtt{X} = \varepsilon\}$.
- $\{\mathtt{X} = w\}$ nil(X) $\{\mathtt{X} = \varepsilon\}$.
- Conditionals C :≡ if top(X) ≡ a [P] are executed if the top symbol on X is a, that is, $\{\mathtt{X}, \vec{\mathtt{Y}} = v\mathtt{a}, \vec{w}\}$ C $\{\mathtt{X}, \vec{\mathtt{X}} = v', \vec{w}'\}$ whenever $\{\mathtt{X}, \vec{\mathtt{Y}} = v\mathtt{a}, \vec{w}\}$ P $\{\mathtt{X}, \vec{\mathtt{X}} = v', \vec{w}'\}$. Otherwise if $\mathtt{a} \not\equiv \mathtt{b}$, then $\{\mathtt{X}, \vec{\mathtt{Y}} = v\mathtt{b}, \vec{w}\}$ C $\{\mathtt{X}, \vec{\mathtt{X}} = v\mathtt{b}, \vec{w}\}$.
- Sequences are executed from the left to the right, that is, if $\{\vec{\mathtt{X}} = \vec{w}\}$ $\mathtt{P}_1$ $\{\vec{\mathtt{X}} = \vec{w}'\}$ and $\{\vec{\mathtt{X}} = \vec{w}'\}$ $\mathtt{P}_2$ $\{\vec{\mathtt{X}} = \vec{w}''\}$, then $\{\vec{\mathtt{X}} = \vec{w}\}$ $\mathtt{P}_1$; $\mathtt{P}_2$ $\{\vec{\mathtt{X}} = \vec{w}''\}$.
- Reading U := X as *copy X to U*, the operational semantics of a loop foreach X [P] is that of the sequence U := X; P; pop(X); ... ; P; pop(X); X := U with $|w|$ occurrences of P; pop(X) whenever w is stored in X before the execution of the loop, where U is some reserved variable that is not allowed to occur elsewhere.

The operational semantics of loop statements follows a *call-by-value semantics* where the contents of the control stack X is saved while providing access to each symbol on X.

We say that an L-program P *computes* a function $f\colon (\Sigma^*)^n \to \Sigma^*$ if P has an *output variable* O and *input variables* $\mathtt{X}_{i_1}, \dots, \mathtt{X}_{i_l}$ among stacks $\mathtt{X}_1, \dots, \mathtt{X}_m$ such that for all $w_1, \dots, w_n \in \Sigma^*$,

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

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

4 The measure μ for L-programs

In the analysis of the computational complexity of L-programs P, the interplay of two kinds of variables will play a major role: the sets $\mathcal{U}(\mathtt{P})$ and $\mathcal{C}(\mathtt{P})$. Intuitively, a variable X is in $\mathcal{U}(\mathtt{P})$ if it occurs as push(a,X) in P and thus might be updated in a run of P, while X is in $\mathcal{C}(\mathtt{P})$ if it controls a loop statement in P. Of course, by the presence of sequence statements these two sets need not be disjoint.

Definition 4.1. The sets $\mathcal{U}(P)$ and $\mathcal{C}(P)$ are inductively defined as follows:

- $\mathcal{U}(\mathtt{imp}) := \mathcal{C}(\mathtt{imp}) := \emptyset$ for each imperative imp, except for $\mathcal{U}(\mathtt{push(a,X)}) := \{\mathtt{X}\}$.
- $\mathcal{U}(\mathtt{P}_1;\ \mathtt{P}_2) := \mathcal{U}(\mathtt{P}_1) \cup \mathcal{U}(\mathtt{P}_2)$ and $\mathcal{C}(\mathtt{P}_1;\ \mathtt{P}_2) := \mathcal{C}(\mathtt{P}_1) \cup \mathcal{C}(\mathtt{P}_2)$.
- $\mathcal{U}(\mathtt{if\ top(X)} \equiv \mathtt{a\ [P]}) := \mathcal{U}(\mathtt{P})$ and $\mathcal{C}(\mathtt{if\ top(X)} \equiv \mathtt{a\ [P]}) := \mathcal{C}(\mathtt{P})$.
- $\mathcal{U}(\mathtt{foreach\ X\ [P]}) := \mathcal{U}(\mathtt{P})$ and

$$\mathcal{C}(\mathtt{foreach\ X\ [P]}) := \begin{cases} \mathcal{C}(\mathtt{P}) \cup \{\mathtt{X}\} & \text{if } \mathcal{U}(\mathtt{P}) \neq \emptyset \\ \mathcal{C}(\mathtt{P}) & \text{else.} \end{cases}$$

Definition 4.2 (Control). Let $\mathtt{P}$ be an L-program. The relation *control in* $\mathtt{P}$, denoted $\overset{\mathtt{P}}{\to}$, is defined as the transitive closure of the following binary relation $\prec_{\mathtt{P}}$ on $\mathcal{V}(\mathtt{P})$:

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

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

For sets of variables V, W we say that W *depends on* V *in* $\mathtt{P}$ if some variable $\mathtt{X} \in V$ controls some variable $\mathtt{Y} \in W$ in $\mathtt{P}$. Accordingly, W *is independent of* V *in* $\mathtt{P}$ if no variable in W is controlled (in $\mathtt{P}$) by a variable in V.

Definition 4.3 (The μ-measure of L-programs). The *μ-measure* of L-programs $\mathtt{P}$, denoted by $\mu(\mathtt{P})$, is inductively defined as follows:

- $\mu(\mathtt{imp}) := 0$ for every imperative $\mathtt{imp}$.
- $\mu(\mathtt{if\ top(X)}\!\equiv\!\mathtt{a\ [P]}) := \mu(\mathtt{P})$.
- If $\mathtt{P}$ is a sequence $\mathtt{P}_1\,;\ldots;\mathtt{P}_n$, then $\mu(\mathtt{P}) := \max\{\mu(\mathtt{P}_1), \ldots, \mu(\mathtt{P}_n)\}$.
- If $\mathtt{P}$ is of the form $\mathtt{foreach\,X\,[Q]}$, then we consider two cases:
  - If $\mathtt{Q}$ is not a sequence, then $\mu(\mathtt{P}) := \mu(\mathtt{Q})$.
  - If $\mathtt{Q}$ is a sequence $\mathtt{Q}_1\,;\ldots;\mathtt{Q}_n$ $(n \geq 2)$ with $k := \mu(\mathtt{Q})$, then

$$\mu(\mathtt{P}) := \begin{cases} k & \text{if each } \mathcal{C}(\mathtt{Q}_i) \text{ with } \mu(\mathtt{Q}_i) = k \\ & \text{is independent of } \mathcal{U}(\mathtt{Q}_i) \text{ in } \mathtt{Q}^{-i} \\ k+1 & \text{else} \end{cases}$$

  where $\mathtt{Q}^{-i}$ denotes the L-program $\mathtt{Q}_1\,;\ldots;\mathtt{Q}_{i-1}\,;\mathtt{Q}_{i+1}\,;\ldots;\mathtt{Q}_n$.

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

Definition 4.4. A sequence $\mathtt{P} :\equiv \mathtt{P}_1\,;\ldots;\mathtt{P}_n$ has a *top circle* if there exists a component $\mathtt{P}_i$ with $\mu(\mathtt{P}_i) = \mu(\mathtt{P})$ such that $\mathcal{C}(\mathtt{P}_i)$ depends on $\mathcal{U}(\mathtt{P}_i)$ in $\mathtt{P}^{-i} :\equiv \mathtt{P}_1\,;\ldots;\mathtt{P}_{i-1}\,;\mathtt{P}_{i+1}\,;\ldots;\mathtt{P}_n$.

By definition 4.4 one can restate the critical case in the definition of the measure μ as:

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

Accordingly, we will show that the polynomial-time computable functions coincide with the functions computable by an L-program where each body of a loop is *circle free*, that is, it has no top circle.

Note that conditionals $\mathtt{if\ X}\!\equiv\!\varepsilon\ \mathtt{[Q]}$ and $\mathtt{if\ X}\!\not\equiv\!\varepsilon\ \mathtt{[Q]}$ with μ-measure $\mu(\mathtt{Q})$ can defined by

$$\begin{aligned} &\mathtt{if\ X}\!\equiv\!\varepsilon\ \mathtt{[Q]} :\equiv \mathtt{nil(U)\,;\,push(a,U)\,;\,foreach\,X\,[pop(U)]\,;\,if\ top(U)}\!\equiv\!\mathtt{a\ [Q]} \\ &\mathtt{if\ X}\!\not\equiv\!\varepsilon\ \mathtt{[Q]} :\equiv \mathtt{if\ top(X)}\!\equiv\!\mathtt{a}_1\ \mathtt{[Q]}\,;\ldots;\mathtt{if\ top(X)}\!\equiv\!\mathtt{a}_l\ \mathtt{[Q]} \end{aligned}$$

where $\mathtt{U}$ is some *new* variable, and $\mathtt{a}$ is an arbitrary letter in $\Sigma := \{\mathtt{a}_1, \ldots, \mathtt{a}_l\}$.

5 The Bounding Theorem

In this section we will show that for every function f computed by an L-program with μ-measure n one can find a *length bound* $b \in \mathcal{E}^{n+2}$, that is, $|f(\vec{w})| \leq b(|\vec{w}|)$ for all $\vec{w}$. It suffices to show this Bounding Theorem for a subclass of programs in L, called *core programs.* The latter comprise those stack manipulations which do contribute to computational complexity of L-programs. The base case is treated separately, showing that every function computed by a core program with μ-measure 0 has a polynomial length bound. To prove the general case, we show that every core program $\mathtt{P}$ with μ-measure $n+1$ has a *length bound* $\mathtt{P}'$ with μ-measure $n+1$. The structure of $\mathtt{P}'$ we call *flattened out* will be such that a straightforward inductive argument shows that every function computed by $\mathtt{P}'$ has a length bound in $\mathcal{E}^{n+2}$.

Definition 5.1 (Core programs). *Core programs* are L-programs built from imperatives `push(a,X)` by sequencing and loop statements.

Note 5.2. *The chosen call-by-value semantics of loops ensures that core programs are* length-monotonic, *i.e. if* $\mathtt{P}$ *is a core program with variables* $\vec{\mathtt{X}}$, *and if* $\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{P}\ \{\vec{\mathtt{X}} = \vec{u}\}$ *and* $\{\vec{\mathtt{X}} = \vec{w}'\}\ \mathtt{P}\ \{\vec{\mathtt{X}} = \vec{u}'\}$ *where* $|\vec{w}| \leq |\vec{w}'|$ *(component-wise), then* $|\vec{u}| \leq |\vec{u}'|$. *Hence every function computed by a core program is length-monotonic, too.*

Lemma 5.3. *For every core program* $\mathtt{P} :\equiv \mathtt{foreach\ X\ [Q]}$ *with* $\mu(\mathtt{P}) = 0$, $\overset{\mathtt{P}}{\rightarrow}$ *is irreflexive.*

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

Lemma 5.4. *Let* $\mathtt{P}$ *be a core program with irreflexive* $\overset{\mathtt{P}}{\rightarrow}$. *Let* $\mathtt{P}$ *have variables among* $\vec{\mathtt{X}} := \mathtt{X}_1, \dots, \mathtt{X}_n$, *and for* $i = 1, \dots, n$, *let* V^i *denote the list of those variables* $\mathtt{X}_j$ *which control* $\mathtt{X}_i$ *in* $\mathtt{P}$. *Then there are polynomials* $p_1(V^1), \dots, p_n(V^n)$ *such that for all* $\vec{w} := w_1, \dots, w_n$,

$$\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{P}\ \{|\mathtt{X}_1| \leq |w_1| + p_1(|\vec{w}^1|), \dots, |\mathtt{X}_n| \leq |w_n| + p_n(|\vec{w}^n|)\}$$

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

Proof. By induction on the structure of core programs $\mathtt{P}$ with irreflexive $\overset{\mathtt{P}}{\rightarrow}$. In the *base case* $\mathtt{P} \equiv \mathtt{push(a,X_1)}$, we know $V^1 = \emptyset$ and hence $p_1 := 1$ will do. As for the *step case*, $\mathtt{P}$ is either of the form $\mathtt{foreach\ X}_j\ \mathtt{[Q]}$ or $\mathtt{P}$ is a sequence $\mathtt{Q}_1; \dots; \mathtt{Q}_l$ $(l \geq 2)$.

Case $\mathtt{P} \equiv \mathtt{foreach\ X}_j\ \mathtt{[Q]}$. The I.H. on $\mathtt{Q}$ yields polynomials $p_1(V^1), \dots, p_n(V^n)$ satisfying

(1) $$\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}\ \{|\mathtt{X}_1| \leq |w_1| + p_1(|\vec{w}^1|), \dots, |\mathtt{X}_n| \leq |w_n| + p_n(|\vec{w}^n|)\}.$$

As $\overset{\mathtt{P}}{\rightarrow}$ is irreflexive, then so is $\overset{\mathtt{Q}}{\rightarrow}$, implying that the relation $\mathtt{X} \sqsubseteq \mathtt{Y} :\Leftrightarrow \mathtt{X} \overset{\mathtt{Q}}{\rightarrow} \mathtt{Y}$ or $\mathtt{X} \equiv \mathtt{Y}$ defines a partial order on $\vec{\mathtt{X}}$. Therefore, we can proceed by induction on $\sqsubseteq$ showing that for every $i = 1, \dots, n$ there is a polynomial $q_i(m, V^i)$ such that for all $m, \vec{w}$,

(2) $$\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{|\mathtt{X}_i| \leq |w_i| + q_i(m, |\vec{w}^i|)\}$$

where $\mathtt{Q}^m$ denotes the sequence $\mathtt{Q};\ldots;\mathtt{Q}$ (m times $\mathtt{Q}$). Note that (2) implies the statement of the lemma for the current case $\mathtt{P} \equiv \mathtt{foreach}\ \mathtt{X}_j\ \mathtt{[Q]}$. To see this, if $\mathtt{X}_i \notin \mathcal{U}(\mathtt{Q})$ then $V^i = \emptyset$ and the execution of $\mathtt{Q}$ does not alter the contents of $\mathtt{X}_i$, hence $p_i := 0$ will do. Otherwise if $\mathtt{X}_i \in \mathcal{U}(\mathtt{Q})$, then $\mathtt{X}_j \in V^i$ and (2) gives $\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{P}\ \{|\mathtt{X}_i| \le |w_i| + q_i(|w_j|, |\vec{w}^i|)\}$ where $w_j \in \vec{w}^i$. As for the proof of (2), if $V^i = \emptyset$ then p_i in (1) is a constant, implying $\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{|\mathtt{X}_i| \le |w_i| + m \cdot p_i\}$. So consider the case where $V^i \ne \emptyset$. The induction hypothesis for each variable in $V^i := \mathtt{X}_{i_1}, \ldots, \mathtt{X}_{i_l}$ provides polynomials $p_{i_1}(m, V^{i_1}), \ldots, p_{i_l}(m, V^{i_l})$ such that for all $m, \vec{w}$,

$$(3) \qquad \{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{|\mathtt{X}_{i_j}| \le |w_{i_j}| + p_{i_j}(m, |\vec{w}^{i_j}|)\} \qquad \text{for } j = 1, \ldots, l.$$

Here V^{i_j} denotes the variables which control $\mathtt{X}_{i_j}$ in $\mathtt{Q}$. As $\overset{\mathtt{Q}}{\to}$ is irreflexive, we conclude

$$(4) \qquad \mathtt{X}_{i_j} \notin V^{i_j} \text{ for } j = 1, \ldots, l, \text{ and } \mathtt{X}_i \notin V^{i_1} \cup \ldots \cup V^{i_l} \subseteq V^i.$$

Hence for a proof of (2) it suffices to show by induction on m that for all $m, \vec{w}$,

$$(5) \quad \{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{|\mathtt{X}_i| \le |w_i| + m \cdot p_i(|w_{i_1}| + p_{i_1}(m, |\vec{w}^{i_1}|), \ldots, |w_{i_l}| + p_{i_l}(m, |\vec{w}^{i_l}|))\}.$$

The *base case* $m = 0$ is obviously true. As for the *step case* $m \to m+1$, given any $\vec{w}$, let $u_{i_1}, \ldots, u_{i_l}$ be the stack contents obtained from (3) such that for $j = 1, \ldots l$,

$$(6) \qquad \{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{\mathtt{X}_{i_j} = u_{i_j}\} \text{ and } |u_{i_j}| \le |w_{i_j}| + p_{i_j}(|\vec{w}^{i_j}|)$$

and let v_i be the stack contents obtained from the side induction hypothesis satisfying

$$(7) \qquad \{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}^m\ \{\mathtt{X}_i = v_i\} \text{ and } |v_i| \le |w_i| + m \cdot p_i(|u_{i_1}|, \ldots, |u_{i_l}|).$$

Whatever the contents of the remaining stacks, by (1) a further execution of $\mathtt{Q}$ gives

$$\{\mathtt{X}_{i_1} = u_{i_1}, \ldots, \mathtt{X}_{i_l} = u_{i_l}, \mathtt{X}_i = v_i\}\ \mathtt{Q}\ \{|\mathtt{X}_i| \le |v_i| + p_i(|u_{i_1}|, \ldots, |u_{i_l}|)\}.$$

This together with (6), (7) and monotonicity of polynomials gives the required estimation $|\mathtt{X}_i| \le |w_i| + (m+1) \cdot p_i(|w_{i_1}| + p_{i_1}(m+1, |\vec{w}^{i_1}|), \ldots, |w_{i_l}| + p_{i_l}(m+1, |\vec{w}^{i_l}|))$, concluding the proof for the case $\mathtt{P} \equiv \mathtt{foreach}\ \mathtt{X}_j\ \mathtt{[Q]}$.

Case $\mathtt{P} \equiv \mathtt{Q}_1;\ldots;\mathtt{Q}_l$ with $l \ge 2$. Since $\overset{\mathtt{P}}{\to}$ is irreflexive, then so is $\overset{\mathtt{Q}_i}{\to}$ for $i = 1, \ldots, l$. Hence the induction hypothesis for each $\mathtt{Q}_i$ provides polynomials $p_1^i(V^1), \ldots, p_n^i(V^n)$ satisfying

$$(8) \qquad \{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{Q}_i\ \{|\mathtt{X}_1| \le |w_1| + p_1^i(|\vec{w}^1|), \ldots, |\mathtt{X}_n| \le |w_n| + p_n^i(|\vec{w}^n|)\}.$$

Due to a situation similar to (4), after at most $n \cdot (l-1)$ compositions we obtain polynomials $p_1(V^1), \ldots, p_n(V^n)$ satisfying $\{\vec{\mathtt{X}} = \vec{w}\}\ \mathtt{P}\ \{|\mathtt{X}_1| \le |w_1| + p_1(|\vec{w}^1|), \ldots, |\mathtt{X}_n| \le |w_n| + p_n(|\vec{w}^n|)\}$. This completes the proof of the lemma. □

Corollary 5.5 (Base Bounding). *For every core program* $\mathtt{P}$ *with* $\mu(\mathtt{P}) = 0$ *and variables* $\vec{\mathtt{X}} := \mathtt{X}_1, \ldots, \mathtt{X}_n$ *one can find polynomials* $p_1(\vec{\mathtt{X}}), \ldots, p_n(\vec{\mathtt{X}})$ *such that for all* $\vec{w} := w_1, \ldots, w_n$,

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

In particular, if $\mathtt{P}$ *computes a function* f*, then* f *has a* polynomial length bound*, that is, a polynomial* p *satisfying* $|f(\vec{w})| \le p(|\vec{w}|)$.

Proof. The statement of the corollary follows from Lemma 5.4, Lemma 5.3, Note 3.2, and the fact that the polynomials are closed under composition. □

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

Definition 5.6. Let $\mathtt{P}$, $\mathtt{Q}$ be core programs such that $\mathcal{V}(\mathtt{P}) := \{\mathtt{X}_1, \dots, \mathtt{X}_k\}$ is a subset of $\mathcal{V}(\mathtt{Q}) := \{\mathtt{X}_1, \dots, \mathtt{X}_k, \mathtt{Y}_1, \dots, \mathtt{Y}_l\}$. We say that $\mathtt{Q}$ is a *length bound* on $\mathtt{P}$, denoted $\mathtt{P} \ll \mathtt{Q}$, if

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

Definition 5.7. We say that a loop $\texttt{foreach X [Q]}$ with μ-measure $n+1$ is *simple* if $\mathtt{Q}$ has μ-measure n. A core program $\mathtt{P} :\equiv \mathtt{P}_1;\dots;\mathtt{P}_k$ with μ-measure $n+1$ is called *flattened out* if each component $\mathtt{P}_i$ is either a simple loop or else $\mu(\mathtt{P}_i) \leq n$.

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

Definition 5.8. The *degree* of a core program $\mathtt{P}$, denoted $\deg(\mathtt{P})$, is inductively defined by:

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

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

Lemma 5.9 (Degree zero). *Every core program $\mathtt{P}$ with μ-measure $n+1$ and degree 0 is flattened out.*

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

Lemma 5.10 (Degree reduction). *For every core program $\mathtt{P} :\equiv \texttt{foreach X [Q]}$ with $\mu(\mathtt{P}) = n{+}1$ and degree > 0 one can find a core program $\mathtt{P}'$ satisfying $\mathtt{P} \ll \mathtt{P}'$, $\mu(\mathtt{P}') = n+1$ and $\deg(\mathtt{P}') < \deg(\mathtt{P})$.*

Proof. Let $\texttt{P} :\equiv \texttt{foreach X [Q]}$ be an arbitrary core program with μ-measure $n+1$ and degree > 0. According to definition 5.8 and Note 3.2, we distinguish two cases on Q.

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

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

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

Case Q is a sequence $\texttt{Q}_0 ; \dots ; \texttt{Q}_k$ without top circle, each component is either an imperative or a loop, and there is a component $\texttt{Q}_i :\equiv \texttt{foreach Y [R]}$ with $\mu(\texttt{Q}_i) = \mu(\texttt{Q}) = n+1$. In this case we define $\texttt{P}' :\equiv \texttt{foreach X [P}_1\texttt{] ; foreach Z [P}_2\texttt{]}$ where

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

for some *new* variable Z and arbitrary letter $\texttt{a} \in \Sigma$. As for $\texttt{P} \ll \texttt{P}'$, let $\#\texttt{R}$ denote the number of times R is executed in a run of P. Since Q has no top circle, Y is independent of $\mathcal{U}(\texttt{R})$ in $\texttt{Q}^{-i} :\equiv \texttt{Q}_0 ; \dots ; \texttt{Q}_{i-1} ; \texttt{Q}_{i+1} ; \dots ; \texttt{Q}_k$. Therefore we obtain $\{\vec{\texttt{X}} = \vec{w}, \texttt{Z} = v\}\ \texttt{P}_1\ \{|\texttt{Z}| \geq \#\texttt{R}\}$, implying $\texttt{P} \ll \texttt{P}'$ by monotonicity of core programs.

It remains to show $\mu(\texttt{P}') = n+1$ and $\deg(\texttt{P}') < \deg(\texttt{P})$. First observe that $\texttt{P}_1$ is a sequence without top circle such that each component has a μ-measure $\leq n+1$. Furthermore, observe that R contains a loop, as $\texttt{Q}_i$ has μ-measure $n+1$. We distinguish two subcases.

Subcase $\texttt{Q}_i$ is a simple loop, that is, $\mu(\texttt{Q}_i) = 1 + \mu(\texttt{R})$ and R is a sequence with a top circle. If $\texttt{Q}_i$ is the only component of Q with μ-measure $n+1$, then $\texttt{P}_2$ is a sequence with a top circle, implying $\mu(\texttt{P}') = \mu(\texttt{foreach Z [P}_2\texttt{]}) = n+1$ and $\deg(\texttt{P}') = \deg(\texttt{foreach X [P}_1\texttt{]})$. As $\texttt{P}_1$ is a sequence without top circle and $\mathrm{nd}(\texttt{Q}_i) \geq 2$, we obtain $\deg(\texttt{foreach X [P}_1\texttt{]}) < \deg(\texttt{P})$. Otherwise if Q has a further component with μ-measure $n+1$, then both $\texttt{P}_1$ and $\texttt{P}_2$ are sequences without top circle, implying $\mu(\texttt{P}') = \mu(\texttt{foreach Z [P}_2\texttt{]}) = n+1$ and, as R contains a loop, $\deg(\texttt{P}') = \deg(\texttt{foreach X [P}_2\texttt{]})$. As $\mathrm{nd}(\texttt{Q}_i) > \mathrm{nd}(\texttt{R})$, we obtain $\deg(\texttt{foreach X [P}_2\texttt{]}) < \deg(\texttt{P})$ as required, concluding the current subcase.

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

Lemma 5.11 (Flattening). *For every core program* P *with* μ*-measure* $n+1$ *one can find a flattened out core program* $\texttt{P}'$ *satisfying* $\texttt{P} \ll \texttt{P}'$ *and* $\mu(\texttt{P}') = n+1$.

Proof. The statement follows from Note 3.2, Lemma 5.10 and Lemma 5.9. □

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

Theorem 5.12 (Bounding). *Every function f computed by an L-program with μ-measure n has a* length bound $b \in \mathcal{E}^{n+2}$ *satisfying* $|f(\vec{w})| \leq b(|\vec{w}|)$.

Proof. It suffices to prove the statement of the theorem for core programs only, since every L-program `P` has a core program `P'` such that $\mu($`P`$) = \mu($`P'`$)$ and `Q` $\ll$ `P'` for every subprogram `Q` of `P`. Just let `P'` result from `P` by simultaneously replacing all occurrences of imperatives `nil(X)` or `pop(X)` with `foreach X [push(b,V)]`, and all conditionals `if top(X)` $\equiv$ `a [Q]` with `foreach X [push(b,V)] ; Q`, for some *new* variable `V` and some letter `b` $\in \Sigma$.

We proceed by induction on n showing the statement of the theorem for core programs. The *base case* $n = 0$ has been shown in Corollary 5.5. As for the *step case* $n \to n+1$, let `P` be an arbitrary core program with μ-measure $n+1$. We apply the Flattening Lemma 5.11 to obtain a core program `P'` of the form $\mathtt{P}_1 ; \ldots ; \mathtt{P}_k$ where each component $\mathtt{P}_i$ is either a simple loop or else $\mu(\mathtt{P}_i) \leq n$, and such that `P` $\ll$ `P'` and $\mu($`P'`$) = n+1$. Thus, by the induction hypothesis and by closure of $\mathcal{E}^{n+3}$ under composition, it suffices to show that every function computable by a simple loop has a length bound in $\mathcal{E}^{n+3}$.

So consider an arbitrary simple loop `P` $:\equiv$ `foreach X [Q]`. Hence $\mu($`Q`$) = n$ and by the induction hypothesis each function h_i computed by `Q` has a length bound $b_i \in \mathcal{E}^{n+2}$. We choose a number $c > 0$ such that $b_i(\vec{x}) \leq E^c_{n+1}(\max(\vec{x}))$ for each bound b_i. Now consider an arbitrary function f_1 computed by `P`. Then f_1, possibly together with other functions $f_2, \ldots, f_m$ computed by `P`, can be defined by *simultaneous string recursion* from functions computed by `Q`, that is,

$$\begin{aligned} f_i(\varepsilon, \vec{w}) &= w_{i_j} \\ f_i(va, \vec{w}) &= h_i(v, \vec{w}, f_1(v, \vec{w}), \ldots, f_m(v, \vec{w})) \quad \text{for } i = 1, \ldots m. \end{aligned}$$

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

6 The Characterisation Theorem

Given any alphabet Σ, a *function over* Σ^* is any k-ary function $f\colon \Sigma^* \times \ldots \times \Sigma^* \to \Sigma^*$. This section is concerned with showing that the functions over Σ^* computable by an L-program with μ-measure n coincide with the functions computable by a Turing machine in time $b(|\vec{w}|)$ for some time bound $b \in \mathcal{E}^{n+2}$. Thus, the polynomial-time computable functions over Σ^* are characterised as those functions f which can be computed by an L-program with μ-measure 0.

Definition 6.1. For $n \geq 0$ let $\mathcal{L}^n$ denote the class of all functions f over Σ^* (for some alphabet Σ) which can be computed by an L-program with μ-measure n.

Definition 6.2. For $n \geq 0$ let $\mathcal{G}^n$ denote the class of all functions f over Σ^* (for some alphabet Σ) which can be computed by a Turing machine in time $b(|\vec{w}|)$ for some $b \in \mathcal{E}^n$.

Observe that $\mathcal{G}^0$ is the class FP of polynomial-time computable functions (over some Σ^*).

Lemma 6.3 (E_{n+1}-Computation). *For every $n \geq 0$ one can find an L-program* $\mathtt{LE}_{n+1}$ *with μ-measure n satisfying* $\{\mathtt{Y} = w\}$ $\mathtt{LE}_{n+1}$ $\{|\mathtt{Y}| = E_{n+1}(|w|)\}$.

Proof. By induction on n, where the *base case* for $E_1(x) = x^2 + 2$ is obvious. As for the *step case*, first recall that $E_{n+2}(x) = E_{n+1}(\ldots E_{n+1}(2) \ldots)$ with x occurrences of E_{n+1}.

Using the induction hypothesis on n, we define $\texttt{LE}_{n+2}$ by:

$$\begin{aligned}\texttt{LE}_{n+2} :\equiv\ & \texttt{nil(U); foreach Y [push(a,U)];}\\ & \texttt{nil(Y); push(a,Y); push(a,Y); foreach U [}\texttt{LE}_{n+1}\texttt{].} \quad \square\end{aligned}$$

Theorem 6.4 (Main Characterisation). *For $n \geq 0$: $\mathcal{L}^n = \mathcal{G}^{n+2}$.*

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

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

Hence there is a Turing machine which simulates P on input $\vec{w}$ in time $q_{\text{time}}(b(|\vec{w}|)) \cdot b(|\vec{w}|)$, concluding the proof of the inclusion "$\subseteq$".

As for "$\supseteq$", let $M := (Q, \Gamma, \Sigma, q_0, \delta)$ be an arbitrary one-tape Turing machine running in time $b(|w|)$ on input w, for some $b \in \mathcal{E}^{n+2}$. We show that the function f_M computed by M can be computed by an L-program P over $\Delta := Q \cup \Gamma \cup \{L, N, R\}$ with μ-measure n, where $\Gamma := \{\texttt{a}_1, \ldots, \texttt{a}_k\}$. Assume that δ consists of *moves* $\text{move}_1, \ldots, \text{move}_l$ where

$$\text{move}_i := (\texttt{q}_i, \texttt{a}_i, \texttt{q}'_i, \texttt{a}'_i, D_i)$$

with $D_i \in \{L, N, R\}$, and we may assume that M does not visit cells left to the input. In simulating M by an L-program, we use the following stacks X, Y, Z, L, R such that for each configuration $\alpha(\texttt{q}, \texttt{a})\beta$ in a run of M on w,

$$(*) \qquad [\![\texttt{L}]\!] = \alpha,\ \text{reverse}([\![\texttt{R}]\!]) = \texttt{a}\beta,\ [\![\texttt{Z}]\!] = \texttt{q}$$

where $[\![\texttt{U}]\!]$ denotes the word stored in stack U. The L-program P with μ-measure n satisfying $\{\texttt{X} = w\}$ P $\{\texttt{R} = f_M(w)\}$ will then have the following form:

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

Recall that there is a constant c satisfying $b(x) \leq E^c_{n+1}(x)$, and by Lemma 6.3 there is an L-program of μ-measure n satisfying $\{\texttt{Y} = w\}$ $\texttt{LE}_{n+1}$ $\{|\texttt{Y}| = E_{n+1}(|w|)\}$. Thus, we obtain $\{\texttt{X} = w\}$ COMPUTE-TIME-BOUND(Y) $\{\texttt{X} = w, |\texttt{Y}| = E^c_{n+1}(|w|)\}$ for the following L-program:

$$\texttt{COMPUTE-TIME-BOUND(Y)} :\equiv \texttt{nil(Y); foreach X [push(a,Y)];}\ \texttt{LE}_{n+1}\texttt{;} \ldots \texttt{;}\ \texttt{LE}_{n+1}$$

with c occurrences of $\texttt{LE}_{n+1}$. According to (*), we initialise L, Z, R as follows:

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

where `REVERSE` is an L-program satisfying $\{\mathtt{X} = w\}$ `REVERSE(X;R)` $\{\mathtt{X} = w, \mathtt{R} = \text{reverse}(w)\}$. `SIMULATE-MOVES` is of the form $\mathtt{MOVE}_1;\dots;\mathtt{MOVE}_k$ where $\mathtt{MOVE}_i$ simulates move_i. For legibility, we use `set(U,a)` for `nil(U); push(a,U)`, `settop(U,a)` for `pop(U); push(a,U)`, and `push(top(L),R)` for `if top(L)` $\equiv \mathtt{a}_1$ `[push(`$\mathtt{a}_1$`,R)];` $\dots$ `; if top(L)` $\equiv \mathtt{a}_k$ `[push(`$\mathtt{a}_k$`,R)]`. According to $D_i = R, L, N$ in $\text{move}_i = (\mathtt{q}_i, \mathtt{a}_i, \mathtt{q}'_i, \mathtt{a}'_i, D_i)$, there are three cases for $\mathtt{MOVE}_i$:

```
(R)  if top(Z) ≡ q_i [if top(R) ≡ a_i [push(a'_i,L); set(Z,q'_i);
                                        if R ≡ ε [push(B,R)]; if R ≢ ε [pop(R)]]].

(L)  if top(Z) ≡ q_i [if top(R) ≡ a_i [settop(R,a'_i); set(Z,q'_i);
                                        if L ≡ ε [push(B,R)];
                                        if L ≢ ε [push(top(L),R); pop(L)]]].

(N)  settop(R,a'_i); set(Z,q'_i)
```

The program `OUTPUT(R;O)` reads out of $[\![\mathtt{R}]\!]$ the result $\mathtt{O} = f_M(w)$, i.e. the maximal initial segment of reverse($[\![\mathtt{R}]\!]$) being a word over Σ. Using obvious implementations of conditionals `if top(R)` $\in \Sigma$ `[Q]` and `if top(R)` $\in \Delta\backslash\Sigma$ `[Q]` with μ-measure $\mu(\mathtt{Q})$, we obtain:

```
OUTPUT(R;O) :≡ nil(O); set(Z,a);
               foreach R [if top(R) ∈ Δ\Σ [nil(Z)];
                          if top(R) ∈ Σ [if top(Z) ≡ a [push(top(R),O)]]]
```

This completes the proof of the Characterisation Theorem. □

Note that the proof of $\mathcal{L}^n \subseteq \mathcal{G}^{n+2}$ does not refer to the functions computed an L-program. Furthermore, recalling property (*) in the proof of $\mathcal{G}^{n+2} \subseteq \mathcal{L}^n$, we obtain for each Turing machine with running time in $\mathcal{E}^{n+2}$ a stack program simulation with μ-measure n by canceling the subprogram `OUTPUT(R;O)`. This gives the main result as stated in the abstract.

Corollary 6.5 (Main). *A Turing machine runs in time $b(n)$ for some function $b \in \mathcal{E}^{n+2}$ if and only if it can be simulated by an L-program with μ-measure n.*

7 Sound, adequate and complete measures

We have presented a purely syntactical method for analysing the impact of nesting loops in L-programs on computational complexity. In particular, the method separates programs running in polynomial time (in the size of the input) from programs running in exponential time. More generally, the method separates uniformly programs with running time in $\mathcal{E}^{n+2}$ from programs with running time in $\mathcal{E}^{n+3}$.

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

Definition 7.1. Assume an arbitrary imperative programming language L and an arbitrary program `P` in L. `P` is *feasible* if every function computed by `P` is in FP. `P` is *honestly feasible* if every subprogram of `P` is feasible. `P` is *dishonestly feasible*, or *dishonest* for short, if `P` is feasible, but not honestly feasible.

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

For honestly feasible programs, every subprogram can be simulated by a Turing machine running in polynomial time. Dishonest programs fall into two groups. One group consists of those programs which only compute functions in FP, but without polynomial running time. The other group consists of programs which run in polynomial time, but some subprograms have non-polynomial running time if executed separately. Typical of the latter group are programs of the form `R; if <test> [Q]` where `R` is a program which runs in polynomial time, `<test>` is a test that always fails, and `Q` is an arbitrary program without polynomial running time.

Dishonest programs somewhat lie about their own computational complexity: They contain computationally redundant code the computational complexity of which dominates that of the whole program. Obviously, we cannot expect to separate (by purely syntactical means) the feasible programs from the non-feasible ones if we take into account dishonest programs. Thus, it seems reasonable to restrict our discussion to the honestly feasible programs, and after all, it is the computational complexity inherent in the code we want to analyze and recognize. But even then, our project is bound to fail.

Definition 7.2. Given any stack programming language L, a *measure on L* is a computable function ν: L-programs $\to \mathbb{N}$.

Definition 7.3. Let L be an arbitrary (reasonable) stack programming language containing the core language defined in section 5, and let ν be a measure on L. The pair (ν, L) is called

- *sound* if every L-program with ν-measure 0 is feasible,
- *complete* if every honestly feasible L-program has ν-measure 0, and
- *adequate* if every function in FP is L-computable with ν-measure 0.

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

Theorem 7.4. *The pair (μ, C) is sound and complete.*

Proof. Soundness follows directly from Corollary 6.5. As for completeness, assume that `P` were an honestly feasible core program with $\mu(\texttt{P}) > 0$. Hence `P` would contain a subprogram `foreach X [Q]` where `Q` is a sequence with a top circle. This implies that each time `Q` is executed at least one stack in `Q` doubles the length of its contents. Thus, `P` would contain a non-feasible subprogram, contradicting the assumption that `P` is honestly feasible. □

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

Theorem 7.5. *Let (ν, L) be a sound and adequate pair. Then (ν, L) is incomplete, that is, there exists an honestly feasible program $\mathtt{P} \in L$ such that $\nu(\mathtt{P}) > 0$.*

Proof. Assume an effective enumeration $\{\rho_i\}_{i\in\omega}$ of the Turing machines with alphabet $\{0, 1\}$. Let n be a fixed natural number. It is well-known that there is a function $f_n \in \mathtt{FP}$ 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 (ν, L) is adequate and sound, there is an honestly feasible program $\mathtt{Q} \in L$ with ν-measure 0 such that

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

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

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

is also in L, where X, V, W are *new* stacks. Now, if ρ_n never halts then `foreach V [push(1,W)]` will never be executed, whatever the inputs to P. Thus, if ρ_n never halts, then P is honestly feasible. In contrast, if ρ_n halts after s steps, say, then part `foreach V [push(1,W)]` and part `foreach W [push(1,V)]` will be executed each time the body of the outermost loop is executed whenever $\mathtt{Y} = y$ with $|y| \geq s$. Each such execution implies that the height of stack V is at least doubled. Thus, if ρ_n eventually halts, then P is not feasible. In other words, P is honestly feasible if and only if ρ_n never halts. As P is effectively constructible from n, we conclude that (ν, L) cannot be complete. For if (ν, L) were complete, then ρ_n would never halt if and only if $\nu(\mathtt{P}) = 0$. 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. □

Notably, Theorem 7.5 relates to Gödel's First Incompleteness Theorem. The latter implies that if a first order language is expressive enough, then there is no algorithm which can identify the true statements of the language. Theorem 7.5 says that when a programming language is sufficiently expressive, then there is no algorithm which can identify the honestly feasible programs of the language.

Acknowledgement. We would like to express our gratitude to Martin Dietzfelbinger for numerous inspiring discussions and for useful comments on various drafts of this paper.

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 (2000) 17–30.

[5] S. Bloch. *Functional characterizations of uniform log-depth and polylog-depth circuit families.* In: Proceedings of the Seventh Annual Structure in Complexity Conference, pp. 193–206. IEEE Computer Society Press, 1992.

[6] Bellantoni S. J. and Niggl K.-H. *Ranking primitive recursions: The low Grzegorczyk classes revisited.* SIAM J. of Comput. 29, No 2, 401-415 (2000).

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

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

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

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

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

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

[15] 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, p. 320–343, Birkhäuser, 1994.

[16] 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.

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

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

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

[20] I. Oitavem. *New Recursive Characterizations of the Elementary Functions and the Functions Computable in Polynomial Space.* Revista Mathematica de la Universidad Complutense de Madrid, volumen 10, número 1 (1997).

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

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

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

Proof Mining in L_1-approximation

Ulrich Kohlenbach Paulo Oliva

BRICS[*]
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark
(Extended Abstract)

1 Introduction

We report on the extraction (presented in [KO01]) of the first effective uniform modulus of uniqueness (this notion is defined in section 2) for best polynomial L_1-approximation of continuous functions from the ineffective proof of uniqueness due to Cheney. The extraction of the modulus is based on the technique of monotone functional interpretation (developed in [Koh96]) and is an instance of the following meta-theorem,[1]

Theorem 1.1 ([Koh93a], theorem 4.1) *Let X, K be $\boldsymbol{PA}^\omega$-definable Polish spaces, K compact and consider a sentence which can be written (when formalized in the language of $\boldsymbol{PA}^\omega$) in the form*

$$A := \forall n \in \mathbb{N}; x \in X; y \in K \; \exists k \in \mathbb{N} \, A_1(n, x, y, k),$$

where A_1 is a purely existential. Then the following rule holds:

$$\begin{cases} \boldsymbol{PA}^\omega + AC_{qf}^{1,0} + WKL \vdash \forall n \in \mathbb{N}; x \in X; y \in K \; \exists k \in \mathbb{N} \, A_1(n,x,y,k) \\ \textit{then one can extract a primitive recursive (in the sense of [Göd58]) term } \Phi \textit{ s.t.} \\ \boldsymbol{HA}^\omega \vdash \forall n \in \mathbb{N}; x \in X; y \in K \exists k \leq \Phi(n,x) \, A_1(n,x,y,k). \end{cases}$$

A crucial feature of the functional Φ above is that it does not depend on the element $y \in K$. It should also be noted that Φ depends *on the representation* of x as an element of X. In the present case X is the space of continuous functions on the unit interval (denoted by $C[0,1]$), and according to the representation of $C[0,1]$ [2] the elements $f \in C[0,1]$ are given together with a modulus of uniform continuity, ω_f,

$$\forall x, y \in [0,1] \forall \varepsilon \in \mathbb{Q}_+^* (|x - y| < \omega_f(\varepsilon) \rightarrow |f(x) - f(y)| < \varepsilon),$$

[*]Basic Research in Computer Science, funded by the Danish National Research Foundation.

[1]$\mathbf{PA}^\omega$ denotes Peano Arithmetic (with extensionality) in all finite types and $\mathbf{HA}^\omega$ its intuitionistic variant.

[2]We have to represent $C[0,1]$ w.r.t. the uniform norm $\|\cdot\|_\infty$ (although we below consider the L_1-norm) since $C[0,1]$ is a Polish space only with respect to the former.

which means that the extracted uniform modulus of uniqueness will depend a priori on such a modulus of uniform continuity of f.

In the theorem above, WKL denotes the non-computational principle "binary ('weak') König's lemma" and $AC^{1,0}_{qf}$ the axiom of choice for quantifier free formulas,

$$AC^{1,0}_{qf} : \forall f^1 \exists x^0 A_{qf}(f,x) \rightarrow \exists F^2 \forall f^1 A_{qf}(f,F(f)).$$

We show below that the uniqueness theorem for L_1-approximation (expressed as a sentence B) can be written in the form A (of meta-theorem 1.1), and the functional Φ guaranteed by the meta-theorem is exactly the uniform modulus of uniqueness. In order to apply theorem 1.1 we just have to show that B can be proved in the system $\mathcal{A} := \mathbf{PA}^\omega + AC^{1,0}_{qf} + WKL$ which can be achieved by showing that Cheney's proof of B can be formalized in $\mathcal{A}$. The latter fact was shown in [Koh90].

Note that our case study is twofold: we test a technique of proof mining (monotone functional interpretation in the present case) with respect to its applicability and feasibility, and at the same time we obtain new results in analysis.

In the next section we introduce some notions from L_1-approximation theory and present the uniform modulus of uniqueness Φ. In section 3 we show how Φ can be used to compute the best L_1-approximation of a given function $f \in C[0,1]$ by polynomials of degree $\leq n$.

2 The uniform modulus of uniqueness

For any given continuous function f on the interval $[0,1]$ (we write $f \in C[0,1]$) the L_1-*norm of* f is defined as,

$$||f||_1 := \int_0^1 |f(x)|\, dx.$$

Given a function $f \in C[0,1]$ we define the L_1-*distance of* f from some subspace $H \subset C[0,1]$ by,

$$dist_1(f,H) := \inf_{p \in H} \|f-p\|_1.$$

By 'the problem of L_1-approximation from H' we mean the problem of finding, for a given $f \in C[0,1]$, an element $p \in H$ such that $\|f-p\|_1 = dist_1(f,H)$. L_1-approximation (or 'approximation in the mean') has been extensively studied in numerical mathematics since 1859 when was first considered by Chebycheff (see [Pin89] for a comprehensive survey). The space of polynomials up to degree n (including n) is denoted by P_n. In 1921, Jackson [Jac21] proved that the problem of L_1-approximation from P_n has a unique solution, i.e. for any given function $f \in C[0,1]$ and for a fixed $n \in \mathbb{N}$ there exists a unique polynomial $p_n \in P_n$ such that

$$\|f-p_n\|_1 = dist_1(f,P_n).$$

In 1965, Cheney (cf. [Che65], [Che66]) simplified (in the sense that he eliminated the use of measure theory) and generalized Jackson's uniqueness proof to arbitrary Haar subspaces[3] of $C[0,1]$. This is the proof we analyze in [KO01].

[3] H is a n-dimensional Haar subspace of $C[0,1]$ if 0 is the only element with n roots. Note that P_n is a Haar subspace of $C[0,1]$ of dimension $n+1$.

The uniqueness of the best L_1-approximation from the space P_n can be written as,

$$\forall f \in C[0,1]; n \in \mathbb{N}; p_1, p_2 \in P_n (\bigwedge_{i=1}^{2} (\|f - p_i\|_1 =_{\mathbb{R}} dist_1(f, P_n)) \to p_1 = p_2).$$

which is equivalent in the system $\mathcal{A}$ to,[4]

$$(*)\ \forall f \in C[0,1]; n \in \mathbb{N}; p_1, p_2 \in P_n; k \in \mathbb{N} \exists d \in \mathbb{N}$$
$$(\bigwedge_{i=1}^{2} (\|f - p_i\|_1 - dist_1(f, P_n) \le 2^{-d}) \to \|p_1 - p_2\|_1 \le 2^{-k}).$$

It is important to note that the best L_1-approximation of any given continuous function $f \in C[0,1]$ from P_n lives in a compact subspace of $C[0,1]$, for instance $K_{f,n} := \{p \in P_n : \|p\|_1 \le 2\|f\|_1\}$, i.e. for a fixed $f \in C[0,1]$ its best L_1-approximation p_n must belong to $K_{f,n}$ for if not we would have $\|p_n\|_1 > 2\|f\|_1$ which, by the triangle inequality for the L_1-norm, implies $\|f - p_n\|_1 > \|f\|_1$, and 0 would be better approximant than p_n, a contradiction. Therefore, $(*)$ can be written as,

$$\forall f \in C[0,1]; n \in \mathbb{N}; p_1, p_2 \in K_{f,n}; k \in \mathbb{N} \exists d \in \mathbb{N}$$
$$(\bigwedge_{i=1}^{2} (\|f - p_i\|_1 - dist_1(f, P_n) \le 2^{-d}) \to \|p_1 - p_2\|_1 < 2^{-k}),$$

which has the exact form of the formula A from theorem 1.1. Now, since Cheney's uniqueness proof can be formalized in $\mathcal{A}$, metatheorem 1.1 guarantees the existence of a functional Φ realizing $\exists d$ depending only on f, n and k, i.e. independent of p_1 and p_2,

$$(**)\ \forall f \in C[0,1]; n \in \mathbb{N}; p_1, p_2 \in K_{f,n}; k \in \mathbb{N}$$
$$(\bigwedge_{i=1}^{2} (\|f - p_i\|_1 - dist_1(f, P_n) \le 2^{-\Phi(f,n,k)}) \to \|p_1 - p_2\|_1 < 2^{-k}).$$

In fact, Φ can be easily extended to a modulus on the whole space P_n. In [Koh93a] and [Koh93b], such functionals were called *uniform modulus of uniqueness.*[5]

Notice that, although the first proof of uniqueness of best L_1-approximation was given already in 1921, no effective uniform modulus of uniqueness had ever been presented before. In [Bjö75] (see also [Bjö79]) Björnestal proved that a uniform modulus of uniqueness having the form $(k+c)+\omega_f(k+c)$ exists, for some constant c depending at most on f and n. Kroo ([Kro78] and [Kro81]) improved this result by showing that c needed only to depend on the modulus of uniform continuity of f (but not on any particular value of f). Moreover, he showed that the k-dependency, i.e. $k + \omega_f(k)$, is optimal. Note, however, that neither Björnestal nor Kroo presented the constant c, which means that only the ε-dependency of the uniform modulus of uniqueness was known before.

By applying the techniques of monotone functional interpretation to Cheney's ineffective uniqueness proof we obtained the following explicit uniform modulus of uniqueness (with $\varepsilon \in \mathbb{Q}^*_+$ instead of 2^{-k}),

[4]It is worth noticing that, according to the representation of real numbers by Cauchy sequences with fixed rate of convergence, in this way, equality between real numbers is expressed by a $\forall$-statement.

[5]Special cases of such moduli have been studied extensively in approximation theory under the heading of "strong uniqueness" (see [BL95]).

Theorem 2.1 *Let*

$$\Phi(\omega, n, \varepsilon) := \min\{\tfrac{c_n\varepsilon}{3^{n+2}(n+1)^{n+1}}, \tfrac{c_n\varepsilon}{2}\omega_n(\tfrac{c_n\varepsilon}{2})\},\ where$$
$$c_n := \frac{\lfloor n/2\rfloor!\lceil n/2\rceil!}{2^{n+3}3^{n^2+2n}(n+1)^{n^2+2n+1}}\ and$$
$$\omega_n(\varepsilon) := \min\{\omega(\tfrac{\varepsilon}{4}), \tfrac{\varepsilon}{40(n+1)^4\lceil\frac{1}{\omega(1)}\rceil}\}.$$

The functional Φ is a uniform modulus of uniqueness for the best L_1-approximation of any function f in $C[0,1]$ (having modulus of uniform continuity ω) from P_n, i.e.

$$\forall n \in \mathbb{N}; p_1, p_2 \in P_n; \varepsilon \in \mathbb{Q}^*_+(\bigwedge_{i=1}^{2}(\|f - p_i\|_1 - dist_1(f, P_n) < \Phi(\omega, n, \varepsilon)) \to \|p_1 - p_2\|_1 \leq \varepsilon).$$

Let $\mathcal{P}(f, n)$ denote the projection operator which assigns to any given function $f \in C[0,1]$ and any $n \in \mathbb{N}$ the best L_1-approximation of $f \in C[0,1]$ from P_n, i.e,

$$\mathcal{P}(f, n) := p_n,\ such\ that\ \|f - p_n\|_1 = dist_1(f, P_n).$$

As a corollary of proposition 5.4 from [Koh93a] and theorem 2.1 above we get,

Theorem 2.2 *Let $\Phi_P(\omega, n, k) := \frac{\Phi(\omega,n,k)}{2}$, Φ as defined in Theorem 2.1. Then, Φ_P is a modulus of pointwise continuity for the operator $\mathcal{P}(f, n)$ for all $f \in C[0,1]$ with modulus of uniform continuity ω, i.e.,*

$$\forall \tilde{f} \in C[0,1]; n \in \mathbb{N}; q \in \mathbb{Q}^*_+(\|f - \tilde{f}\|_1 < \Phi_P(\omega, n, \varepsilon) \to \|\mathcal{P}(f, n) - \mathcal{P}(\tilde{f}, n)\|_1 \leq \varepsilon).$$

3 Computing the best L_1-approximation

Now we present one of the applications of the uniform modulus of uniqueness. First we define another norm. Let $a := (a_0, a_1, \ldots, a_n)$ be an $(n+1)$-dimensional tuple of real numbers. The *max-norm of* a is defined as,

$$\|a\|_{max} := \max\{|a_0|, \ldots, |a_n|\}.$$

Since the polynomials of P_n can be viewed as a $(n+1)$-tuples of real numbers (taking the coefficients as the elements of the tuple) it makes sense to speak about the max-norm of $p \in P_n$. In the same way we treat $(n+1)$-tuples of rational numbers as elements of P_n.

Definition 3.1 *An operator $B_f : \mathbb{N} \times \mathbb{N} \to \mathbb{Q}^{n+1}$ computes the sequence of unique best L_1-approximations, $(p_n)_{n\in\mathbb{N}}$, of a function $f \in C[0,1]$ from P_n if for any given $n, k \in \mathbb{N}$ it generates an $(n+1)$-tuple of rationals $B_f(n,k)$ (which can be viewed as a polynomial with rational coefficients in P_n) such that $\|p_n - B_f(n,k)\|_{max} < 2^{-k}$.*

Let $f \in C[0,1]$ be fixed and assume p_n is the best L_1-approximation of f from P_n. Taking p_2 to be p_n in $(**)$ we have,

$$(1)\ \forall f \in C[0,1]; n \in \mathbb{N}; p \in P_n; k \in \mathbb{N} \\ (\|f-p\|_1 - dist_1(f,P_n) \leq 2^{-\Phi(f,n,k)} \to \|p-p_n\|_1 < 2^{-k}).$$

In order to define B_f we need, besides the uniform modulus of uniqueness Φ, a functional $\Psi(f,n,k)$ such that,

$$(2)\ \forall f \in C[0,1]; n,k \in \mathbb{N}(\Psi(f,n,k) \in \mathbb{Q}^{n+1} \wedge \|f-\Psi(f,n,k)\|_1 < dist_1(f,P_n) + 2^{-k}),$$

and a function $\Theta(n,k)$ satisfying,

$$(3)\ \forall n \in \mathbb{N}; p \in P_n; k \in \mathbb{N}(\|p\|_1 < 2^{-\Theta(n,k)} \to \|p\|_{max} < 2^{-k}).$$

It is clear now, by (1), (2) and (3), that the functional defined by

$$B_f(k,n) := \Psi(f,n,\Phi(f,n,\Theta(n,k)))$$

generates the double sequence $(p_{n,k})_{n,k \in \mathbb{N}}$. Note that, if the modulus of uniqueness Φ depended on p_1 and p_2, the functional $B_f(k,n)$ could not be defined, since in order to get (1) we would need to have p_n already.

It is only left to show how the functionals Ψ and Θ can be defined. We sketch here only the definition of $\Psi(f,n,k)$: For $f \in C[0,1]$ and $n,k \in \mathbb{N}$ fixed, we show how to find a polynomial $p \in P_n$ such that $\|f-p\|_1 < dist_1(f,P_n) + 2^{-k}$. As it was pointed out above, such p lives in the compact space $K_{f,n}$. The idea is to build a finite net of elements containing some elements of $K_{f,n}$ (called k-net) such that at least one of them (say p') is $(k+1)$-*close* to p_n, i.e. $\|p_n - p'\|_1 \leq 2^{-k-1}$, which by triangle inequality ($\|f-p'\|_1 \leq \|f-p_n\|_1 + \|p_n - p'\|_1$) yields $(*)$ $\|f-p'\|_1 \leq dist_1(f,P_n) + 2^{-k-1}$. Once we have the k-net we compute $\|f-p\|_1$ for any element p in the net with $(k+1)$-precision and we take a $\tilde{p}$ which gives the minimum value (e.g. the one with smallest code). Since the L_1-norm was computed with $(k+1)$-precision, by $(*)$, we have, $\|f-\tilde{p}\|_1 \leq dist_1(f,P_n) + 2^{-k}$.

This important application of moduli of uniqueness for the computation of unique solutions of existence statements was first investigated and used in [Koh93a]. The computational complexity (in the sense of [Ko86]) of generating the best L_1-approximation of a given $f \in C[0,1]$ from P_n will be analyzed in [Oli].

References

[Bjö75] B.O. Björnestal. Continuity of the metric projection operator i-iii. *The preprint series of Department of Mathematics. Royal Institute of Technology. Stockholm, TRITA-MAT*, 17, 1975.

[Bjö79] B.O. Björnestal. Local Lipschitz continuity of the metric projection operator. *Approximation theory. In: Papers, VIth Semester, Stefan Banach Internat. Math. Center, Warsaw, 1975)*, pages 43–53, 1979.

[BL95] M. Bartelt and W. Li. Error estimates and Lipschitz constants for best approximation in continuous function spaces. *Computers and Mathematics with Application*, 30(3-6):255–268, 1995.

[Che65] E.W. Cheney. An elementary proof of Jackson's theorem on mean-approximation. *Mathematics Magazine*, 38:189–191, 1965.

[Che66] E.W. Cheney. *Approximation Theory.* AMS Chelsea Publishing, 1966.

[Göd58] K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, 12:280–287, 1958.

[Jac21] D. Jackson. Note on a class of polynomials of approximation. *Transactions of the American Mathematical Society*, 22:320–326, 1921.

[Ko86] K.-I. Ko. On the computational complexity of best Chebycheff approximation. *Journal of Complexity*, 2:95–120, 1986.

[KO01] U. Kohlenbach and P. Oliva. Effective bounds on strong unicity in L_1-approximation. BRICS Report Series RS-01-14 (38 pages), BRICS, 2001.

[Koh90] U. Kohlenbach. *Theory of majorizable and continuous functionals and their use for the extraction of bounds from non-constructive proofs: effective moduli of uniqueness for best approximations from ineffective proofs of uniqueness (german).* PhD thesis, Frankfurt, pp. xxii+278, 1990.

[Koh93a] U. Kohlenbach. Effective moduli from ineffective uniqueness proofs. An unwinding of de La Vallée Poussin's proof for Chebycheff approximation. *Annals of Pure and Applied Logic*, 64:27–94, 1993.

[Koh93b] U. Kohlenbach. New effective moduli of uniqueness and uniform a–priori estimates for constants of strong unicity by logical analysis of known proofs in best approximation theory. *Numerical Functional Analysis and Optimization*, 14:581–606, 1993.

[Koh96] U. Kohlenbach. Analysing proofs in Analysis. In W. Hodges, M. Hyland, C. Steinhorn, and J. Truss, editors, *Logic: from Foundations to Applications*, pages 225–260. European Logic Colloquium (Keele, 1993), Oxford University Press, 1996.

[Kro78] A. Kroo. On the continuity of best approximations in the space of integrable functions. *Acta Mathematica Academiae Scientiarum Hungaricae*, 32:331–348, 1978.

[Kro81] A. Kroo. On strong unicity of L_1-approximation. *Proceedings of the American Mathematical Society*, 83(4), 1981.

[Oli] P. Oliva. On the computational complexity of best L_1-approximation. In preparation.

[Pin89] A. Pinkus. *On L_1-Approximation*, volume 93 of *Cambridge Trats in Mathematics.* Cambridge University Press, 1989.

On Type-2 Complexity Classes

Preliminary Report

Chung-Chih Li[*] James S. Royer[*]

15 March 2001

Abstract

There are now a number of things called "higher-type complexity classes." The most promenade of these is the class of *basic feasible functionals* [CU93, CK90], a fairly conservative higher-type analogue the (type-1) polynomial-time computable functions. There is however currently no satisfactory general notion of what a higher-type complexity class should be. In this paper we propose one such notion for type-2 functionals and begin an investigation of its properties. The most striking difference between our type-2 complexity classes and their type-1 counterparts is that, because of topological constrains, the type-2 classes have a much more ridged structure. *Example:* It follows from McCreight and Meyer's Union Theorem [MM69] that the (type-1) polynomial-time computable functions form a complexity class (in the strict sense of Definition 1 below). The analogous result *fails* for the class of type-2 basic feasible functionals.

§*1. Introduction*

Constable [Con73] was one of the first to study the computational complexity of higher-type functionals. In that 1973 paper, he raised two good questions:

1. What is the type-2 analogue of the polynomial-time computable functions?
2. What is the computational complexity theory of the type-2 effectively continuous functionals?

In the years since there has been a fair amount of attention given to addressing the first question, but hardly any to the second. We think that after nearly three decades the man deserves an answer. Herein we make a start at providing one.

[*]Dept. of Elec. Eng. & Computer Science; Syracuse University; Syracuse, NY 13244 USA. Research supported in part by NSF grant CCR-9522987.

Professor Constable will have to wait a bit longer for a full answer to his question because what he seems to have had in mind in 1973 and what we study below are different in a couple of respects. First, Constable was interested in effectively continuous functionals [Odi89] of the general type $(\mathrm{N} \rightharpoonup \mathrm{N})^m \times \mathrm{N}^n \rightharpoonup \mathrm{N}$.[1] We instead focus on partial recursive functionals [Odi89] of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \rightharpoonup \mathrm{N}$ — a much more tractable setting. Second, Constable wanted an extension of Blum's complexity measure axioms [Blu67, Odi99] to type-2 and included an interesting proposal along those lines. At present type-2 computational complexity is at so poorly understood that we believe that concrete, worked-out examples are what is needed. So instead of trying to develop an axiomatic treatment, we follow an approach similar to that of Hartmanis and Stearns [HS65] in studying the complexity properties of a simple, standard model of computation. In our case, the model is the deterministic, multi-tape, oracle Turing machine with Kapron and Cook's answer-length cost convention (more on this shortly).

Outline. The next section sketches a few facts about type-1 complexity theory and explains our focus on complexity classes. Section 3 introduces our model of type-2 computation and some associated notions. Section 4 considers the nature of type-2 time bounds and Section 5 concerns what it means for a type-2 time bound to hold almost everywhere. Section 6 introduces our definition of a type-2 complexity class and presents few elementary results about these classes. Unions of complexity classes and whether these unions are themselves complexity classes are considered in Section 7. Section 8 studies the problem of whether there is a uniformly way to expand a given complexity class to a strictly larger class. Finally, Section 9 contains our conclusions and suggestions for future work.

§2. *A glance at type-1 computational complexity*

Our study of type-2 computational complexity proceeds by rough analogy with the type-1 theory. Thus before considering the situation at type-2, we start by recalling a few basic facts about the type-1 theory.

Our model of computation. Following hoary complexity-theoretic tradition, we take deterministic, multi-tape Turing machines (TMs) as our default model of type-1 computation. Each step of a TM has unit cost. To simplify matters a bit, we also follow the common convention of requiring that each TM must read its entire input string. This forces a TM to have distinct computations on distinct inputs. (We will return to this point later.)

Strings and numbers. Each $x \in \mathrm{N}$ is identified with its dyadic representation over $\{\mathbf{0}, \mathbf{1}\}$. Thus, $0 \equiv \epsilon$, $1 \equiv \mathbf{0}$, $2 \equiv \mathbf{1}$, $3 \equiv \mathbf{00}$, etc. For each $x \in \mathrm{N}$,

[1] *Notation:* N denotes the set of natural numbers and $A \rightharpoonup B$ (respectively, $A \to B$) denotes the collection of all partial (respectively, total) set-theoretic functions from A to B.

$|x|$ denotes the length of its dyadic representation. We will freely pun between $x \in \mathrm{N}$ as a number and a **0**-**1**-string. TMs are thought of computing partial functions over N ($\cong \{\,\mathbf{0},\mathbf{1}\,\}^*$).

The standard indexing and complexity measure. $\mathcal{PR}$ and $\mathcal{R}$ respectively denote the partial recursive and total recursive functions over N. Let $\langle \varphi_i \rangle_{i\in \mathrm{N}}$ be an acceptable indexing [Rog67] of $\mathcal{PR}$ based on TMs. We call i a φ-program for φ_i. For each i and x, let $\Phi_i(x)$ denote the run time of the TM encoded by i on input x. Note that $\langle \Phi_i \rangle_{i\in \mathrm{N}}$ satisfies Blum's complexity measure axioms: (i) $\{\,(i,x) \mid \varphi_i(x)\!\downarrow\,\} = \{\,(i,x) \mid \Phi_i(x)\!\downarrow\,\}$ and (ii) $\{\,(i,x,n) \mid \Phi_i(x) \le n\,\}$ is decidable. Also note that it follows from our requirement that a TM must read all of its input that $|x|+1 \le \Phi_i(x)$ for each i and x.

Ordering on functions and almost everywhere relations. For $f,g\colon A \to B$, $f \le g$ means that for all $x \in A$, $f(x) \le g(x)$; $f < g$, and so on, are defined analogously. For $f,g\colon \mathrm{N} \to \mathrm{N}$, $f =^* g$ means that $\{\,x \mid f(x) = g(x)\,\}$ is co-finite; $f <^* g$, and so on, are defined analogously.

Definition 1 (Type-1 complexity classes). For each $t \in \mathcal{R}$:

$$C(t) \;\;=_{\text{def}}\;\; \{\,\varphi_i \in \mathcal{R} \mid i \in \mathrm{N} \;\&\; \Phi_i \le^* t\,\}. \tag{1}$$

We call $C(t)$ the *complexity class* named by t. ◇

Equation (1) is the standard definition of a complexity class relative to an arbitrary complexity measure Φ. By using special properties of our particular choice of Φ, we could replace the $\le^*$ in (1) with $\le$ and definite essentially the same notion. However, we retain the $\le^*$ as a pedagogical reminder that membership in a complexity class depends on the *asymptotic* behavior of witnessing programs. For example, under many models of type-1 computation one can effectively patch programs so that on some specified finite set of arguments, the complexity is essentially anything you choose and off that finite set of arguments, the complexity is unchanged from the original. Thus, the "inherent complexity" of a program is only revealed in its asymptotic behavior.[2] One consequence of (1) is that to establish $f \notin C(t)$ for given f and t, one must prove that any program for f must have complexity that is infinitely often greater than t. Here is a sample argument along these lines.

Theorem 2 (Rabin [Rab60]). *Suppose $t \in \mathcal{R}$. Then there is an 0–1-valued element $f \in \mathcal{R}$ such that $f \notin C(t)$.*

Proof Sketch. The proof uses a standard cancellation constructions. In the program for f given in Figure 1, C_w = programs cancelled on inputs $< w$ and

[2]Another reason stems from recursive relatedness [Blu67, Odi99]; when you abstract away from a particular model of computation, the almost everywhere bounds are a necessary part of most theorems in the general theory.

```
Input $x$.
$C_0 \leftarrow \emptyset$.
For $w \leftarrow 0$ to $x$ do:
  $S_w \leftarrow \{\, k \le w \mid k \notin C_w \;\&\; \Phi_k(w) \le t(w) \,\}$.
  If $S_w \neq \emptyset$ then $C_{w+1} \leftarrow C_w \cup \{\, \min(S_w) \,\}$ else $C_{w+1} \leftarrow C_w$.
If $S_x = \emptyset$ then return 0 else return $1 \dot{-} \varphi_e(x)$, where $e = \min(S_x)$.
```

Figure 1: The program for f

S_w = the candidates for cancellation on input w. A program i is *cancelled* on input w if and only if $i \in C_{w+1} - C_w$, in which case the construction will guarantee that $\Phi_i(w) \le t(w)$, $f(w) \neq \varphi_i(w)$, and i will never be cancelled again.

It is clear from the program that cancellation works as advertised and f is a 0-1 element of $\mathcal{R}$. It remains to show $f \notin C(t)$. Suppose that i is such that $\Phi_i \le^* t$. Choose $w_0 \ge i$ so that (a) $\Phi_i(w_0) \le t(w_0)$ and (b) for each $k < i$ that is ever cancelled, k has been cancelled before input w_0. Hence, either i has been cancelled on a $w < w_0$ or the construction must cancel i on input w_0. In either case $\varphi_i \neq f$. Therefore, $f \notin C(t)$. □

Honesty. Note that Definition 1 says nothing about the complexity of computing t itself. This is an important issue that is usually dealt with through the notions of honesty and time constructibility.

Definition 3. Suppose $f, g \colon \mathrm{N} \to \mathrm{N}$.

(a) We say that f is *g-honest* if and only if for some i, a φ-program for f, $\Phi_i \le g \circ f$.

(b) We say that f is *honest* if and only if f is g-honest for some $g \in \mathcal{R}$.

(c) We say that f is *time constructible* if and only if $f = \Phi_i$ for some i with $\varphi_i \in \mathcal{R}$. ◇

Intuitively, f is honest provided there is a way of computing f such that the size of each output is roughly commensurate with the time it takes to produce that output. The construction for Rabin's Theorem produces highly dishonest functions. On the other hand, a time constructible function is as honest as it is possible to be. Roughly, honest functions provide good names for complexity classes, whereas complexity classes named by dishonest functions can be quite pathological (see Theorems 23 and 24 below). In this paper we will not deal directly with type-2 analogues of honesty and time constructibility. However, they will be important background concerns, and we will see that there is some amount of honesty built into our notion of type-2 time bound.

Why are complexity classes of interest? One of the central obsessions of computation complexity theory is the attempt to draw sharp boundaries between the computationally feasible and infeasible, where the notions of feasible and infeasible vary with context. Given an arbitrary $t \in \mathcal{R}$, $C(t)$ is unlikely to represent anyone's notion of feasibility. However, $C(t)$ is a very simple and elegant way of representing a complexity-theoretic boundary. Thus, if you want to understand computational complexity, one of the first things you want to study is the nature of these boundaries.

This is enough about the type-1 theory for the moment. Our goal in the next few sections is to introduce a sensible type-2 analogue of Definition 1.

§3. *Type-2 computations and their costs*

For our default model of type-2 computation we take deterministic, multi-tape *oracle Turing machines* (OTMs). Under our setup OTMs are TMs that are augmented with two special tapes: a *query tape* and a *reply tape*, and one special instruction: *query*. To make a query of an oracle $f\colon \mathrm{N} \to \mathrm{N}$, an OTM writes a **0-1** string (interpreted as the dyadic representation of an $x \in \mathrm{N}$) on the query tape and goes into its query state, whereupon the contents of the query tape are erased and the contents of the reply tape become the dyadic representation of $f(x)$. We also require that each OTM must read all of its type-0 input before halting, and additionally, that immediately after making the query, an OTM must read all of the answer to said query. Each step of an OTM has unit cost, but our requirement that OTMs read all of each oracle response makes our cost model equivalent to Kapron and Cook's *answer-length cost model* [KC96].

Why OTMs? No one outside of complexity theory cares much for TMs or OTMs as models of computation. So our use of OTMs is a poor marketing choice. In their favor, OTMs are a simple, conservative model of computation with a simple, conservation notion of cost. Hence, reasoning about their complexity is straightforward (or as straightforward as reasoning about complexity can ever be) and this just what we want from our model of computation in our initial foray into this territory. Extending our results to other basic models of computation should not be that hard, but we need to know the general shape of the results first.

The unit cost model for OTMs. If we drop the requirement that each OTM must read the entire answer to each query, then we obtain the *unit cost model* for OTMs. Working with this model is more difficult than the answer-length cost model and the results tend to be weaker. This unit cost model is studied in [Li01], but we shall not discuss it further in this paper.

Finite functions. Let $\mathcal{F}$ denote the collection of finite functions over N, i.e., each $\sigma\colon \mathrm{N} \rightharpoonup \mathrm{N}$ is defined on only finitely many arguments. In the following σ

and τ (with or without decorations) range over $\mathcal{F}$. We identify each σ with its graph: $\{ (x, \sigma(x)) \mid \sigma(x)\downarrow \}$. We shall assume some canonical representation of the elements of $\mathcal{F}$ and typically treat them as type-0 arguments to functions. For each σ, define $\overline{\sigma}\colon \mathrm{N} \to \mathrm{N}$ by:

$$\overline{\sigma}(x) \quad = \quad \begin{cases} \sigma(x), & \text{if } \sigma(x)\downarrow; \\ 0, & \text{otherwise.} \end{cases}$$

The standard indexing and complexity measure. The class of functionals computed the OTMs sketched above are called the *partial recursive functionals* (of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \rightharpoonup \mathrm{N}$) in Odifreddi [Odi89]. We denote this class by $\mathcal{PRF}$ and the total members of this class by $\mathcal{RF}$. Let $\langle \boldsymbol{\varphi}_i \rangle_{i \in \mathrm{N}}$ be an acceptable indexing of $\mathcal{PRF}$ based on OTMs. We call i a $\boldsymbol{\varphi}$-program for $\boldsymbol{\varphi}_i$. For each i, f, and x, let $\boldsymbol{\Phi}_i(f, x)$ denote the run time of the OTM encoded by i on input (f, x). Note that it follows from our requirement that an OTM must read all of its input that $|x|+1 \leq \boldsymbol{\Phi}_i(f, x)$ for each i, f, and x. For each i, f, x, and n, define $Q_i(f, x, n) =$ the set of queries issued in the first $\min(n, \boldsymbol{\Phi}_i(f, x))$ steps of the computation of $\boldsymbol{\varphi}$-program i on input (f, x), $Q_i(f, x) = \cup_n Q_i(f, x, n)$, $\mathrm{Use}_i(f, x, n) = \{ (x, f(x)) : x \in Q_i(f, x, n) \}$, and $\mathrm{Use}_i(f, x) = \cup_n \mathrm{Use}_i(f, x, n)$. Also, for each i, σ, and x, we define

$$\boldsymbol{\Phi}_i(\sigma, x) \quad = \quad \begin{cases} \boldsymbol{\Phi}_i(\overline{\sigma}, x), & \text{if } Q_i(\overline{\sigma}, x) \subseteq \{ y \mid \sigma(y)\downarrow \}; \\ n, & \text{otherwise, where } n \text{ is the number of steps} \\ & \quad \text{taken up to the issuance of the first query} \\ & \quad \text{“}\sigma(y) =?\text{” where } \sigma(y)\uparrow. \end{cases}$$

§4. *Type-2 time bounds*

Our current goal is to lift Definition 1 to type-level 2 in a reasonable way. The key issue in this is what should be the type-2 translation of the inequality $\Phi_i \leq^* t$ of (1). In place of Φ_i we clearly should use $\boldsymbol{\Phi}_i$, but there are two harder questions:

1. What should stand in place of $\leq^*$?

2. What should stand in place of t?

We examine the first question in the next section. Here we shall consider how to sensibly express time bounds on type-2 computations.

What we don't do, and why. One way to proceed is to use arbitrary elements of $\mathcal{RF}$ as time bounds. That is, given $T \in \mathcal{RF}$, we could say that $\boldsymbol{\varphi}$-program i has complexity everywhere bounded by T if and only if $\boldsymbol{\Phi}_i \leq T$ (i.e., for all f and x, $\boldsymbol{\Phi}_i(f, x) \leq T(f, x)$). Something of this sort is briefly considered by Kapron [Kap91] and Seth [Set94]. This sort of bound has the following troublesome feature.

Proposition 4. *Suppose that $\mathbf{\Phi}_i \leq T$ and that b is some $\boldsymbol{\varphi}$-program b for T. Then, for all f and x, $Q_i(f,x) \subseteq Q_b(f,x)$.*

Proof. Suppose by way of contradiction that $y \in (Q_i(f,x) - Q_b(f,x))$ for some particular f and x. Let f' be such that $f'(z) = f(z)$ for $z \neq y$ and $f'(y) = 2^{1+T(f,x)}$. Then $T(f',x) = T(f,x)$ since $\boldsymbol{\varphi}$-program b fails to query f on y. Moreover, $\mathbf{\Phi}_i(f',x) > T(f,x)$ since $\boldsymbol{\varphi}$-program i on input (f',x) will query f' on y and the cost of this query is greater than $T(f,x)$. Therefore, $\mathbf{\Phi}_i(f',x) > T(f,x) = T(f',x)$, a contradiction. □

Thus, for $\mathbf{\Phi}_i \leq T$ to hold it must be the case that for any $\boldsymbol{\varphi}$-program b for T and any input (f,x), the $\boldsymbol{\varphi}$-program b must anticipate all of the possible questions the computation of i on (f,x) might ask and ask them itself. This seems like an odd thing for a humble bound to do. In particular, if T is honest and small (in some reasonable senses), then $\{\, \boldsymbol{\varphi}_i \in \mathcal{RF} \mid i \in \mathrm{N} \;\&\; \mathbf{\Phi}_i \leq T \,\}$ must be very restricted since the set of queries a $\boldsymbol{\varphi}_i$ in this collection will be quite circumscribed.

Our approach. To avoid having our bounding functionals issue queries, we make them a particular sort of enumeration operator [Rog67, Odi89]. That is, the bounding functionals can be thought of as passive observers of the computations they are set to bound; at any point of the computation, the bounding functional will have "bounding value" based on what the functional has seen of the input and the queries. This scheme is directly inspired by the standard clocking scheme for second-order polynomially bounded OTMs [KC96, Set92]. We proceed formally as follows.

Definition 5. Suppose $\beta\colon \mathcal{F} \times \mathrm{N} \to \mathrm{N}$ is total computable.

(a) We say that β determines a *weak type-2 time bound* if and only if it satisfies the following three conditions, for all f, σ, and x,

Nontriviality: $\beta(\sigma,x) \geq |x|+1$.

Convergence: $\lim_{\tau\to f} \beta(\tau,x)\!\downarrow\, < \infty$.

Boundedness: $\sup_{\tau\subset f} \beta(\tau,x) = \lim_{\tau\to f} \beta(\tau,x)$.

Let WB be the collection of all such β's.

(b) We say that β determines a *strong type-2 time bound* if and only if β satisfies the nontriviality and convergence conditions as above as well as satisfying, for all σ, σ', and x:

Monotonicity: $\sigma \subseteq \sigma'$ implies $\beta(\sigma,x) \leq \beta(\sigma',x)$.

Let SB be the collection of all such β's. ◇

Clearly, if SB $\subset$ WB. Unless we say otherwise, β will denote an element of WB in the following.

DEFINITION 6.

(a) We say that the run time of φ-program i on input (f, x) is *bounded* by β (written $\varphi_{i,\beta}(f,x)\Downarrow$) if and only if, for each n, $\Phi(i, \sigma_n, x) \leq \beta(\sigma_n, x)$, where $\sigma_n = \text{Use}_i(f, x, n)$.

(b) We say that the computation of φ-program i on input (f, x) is *clipped* by β (written $\varphi_{i,\beta}(f,x)\Uparrow$) if and only if not $\varphi_{i,\beta}(f,x)\Downarrow$.

(c) Define $E_{i,\beta} = \{\, (f, x) \mid \varphi_{i,\beta}(f,x)\Uparrow \,\}$; we call $E_{i,\beta}$ the *exception set* for i and β.

(d) We say that the run time of φ-program i is *everywhere bounded* by β if and only if $E_{i,\beta}$ is empty. $\diamond$

EXAMPLE 7.

(a) Suppose $\varphi_i \in \mathcal{RF}$ and, for each σ and x, let $\beta(\sigma, x) = \mathbf{\Phi}_i(\sigma, x)$. Then $\beta \in$ SB and it is no surprise that the run time of φ-program i is everywhere bounded by β.

(b) For each a, k, d, x, and σ, define $\beta_{a,k,0}(\sigma, x) = a \cdot (|x| + 1)^k$ and also $\beta_{a,k,d+1}(\sigma, x) = a \cdot (|w| + |x| + 1)^k$, where $w = \max(\{\, \sigma(y) \mid |y| \leq \beta_{a,k,d}(\sigma, x) \;\;\&\;\; \sigma(y)\downarrow \,\})$. Then each $\beta_{a,k,d} \in$ SB and the class of BFFs of type $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \to \mathrm{N}$ is exactly $\bigcup_{a,k,d} \{\, \varphi_i \mid$ the run time of φ-program i is everywhere bounded by $\beta_{a,k,d} \,\}$ [Set92, IKR01]. $\diamond$

§5. *Type-2 almost everywhere bounds*

We want to speak of the run time of φ-program i being *almost everywhere* bounded by β. Intuitively, this should mean that in some appropriate sense $E_{i,\beta}$ is finite. In the realm of function spaces, "finite" usually corresponds to compact in some topology. So the question of almost everywhere bounds comes down to a choice of topology.

What we don't do, and why. Our OTMs compute over $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N}$. That space is isomorphic to N^{N} which has a well-known topology due to Baire. Let $\mathcal{B}$ denote this topology on $(\mathrm{N} \to \mathrm{N}) \times \mathrm{N}$. For $\mathcal{B}$, it suffices to take $\{\, \langle\!\langle \sigma, x \rangle\!\rangle \mid \sigma \in \mathcal{F},\ x \in \mathrm{N} \,\}$ as the collection of basic open sets, where for each σ and x,

$$\langle\!\langle \sigma, x \rangle\!\rangle \;\;=_{\text{def}}\;\; \{\, (f, x) \mid f \supset \sigma \,\}.$$

The problem with $\mathcal{B}$ is that the compact sets are all too small for our purposes. This is shown by:

PROPOSITION 8. *If an $E_{i,\beta}$ is $\mathcal{B}$-compact, then this $E_{i,\beta}$ is empty.*

PROOF. It follows from Definitions 5(a) and 6(c) that $E_{i,\beta}$ is open in $\mathcal{B}$. But the only open compact set in this topology is $\emptyset$. $\square$

Roughly, the more open sets one has in a topology, the more restricted are its compact sets. $\mathcal{B}$ and "large" topologies in general thus fail to provide sufficiently large compact sets so as to obtain nontrivial almost everywhere relations.

Our approach. To address the problem of what topology to use, we shift our attention from the set of possible inputs of an OTM to the set of possible *computations* of an OTM. To motivate this shift, let us first consider a particular ordinary TM M that ignores the convention about reading its entire input. This M acts as follows:

> Upon staring, M examines the first symbol on the input tape. If this is a $\mathbf{0}$, M immediately halts with output $\mathbf{0}$; otherwise, M reads the rest of the input and then halts with output $\mathbf{1}$.

Clearly there are infinitely many inputs on which M halts with output $\mathbf{0}$. However, there is only one computation of M that produces output $\mathbf{0}$: on an input of the form $\mathbf{0}\{\,\mathbf{0},\mathbf{1}\,\}^*$ the machine never looks beyond the initial $\mathbf{0}$, hence all such inputs produce the same computation. Therefore if we want to say that such an M does something for all but finitely many cases, we must specify whether cases in question are inputs or computations — each choice has its own faults and merits.

Now, any halting computation of an OTM has $2^{\aleph_0}$-many inputs which produce that computation. So if we want to say that OTM does something for all but finitely many cases, we again must choose between these cases corresponding to inputs or computations. $\mathcal{B}$ roughly corresponds to the "inputs" choice. No single topology corresponds to the "computations" choice, but we need not be restricted to a single topology. The next two definitions introduce several topologies we need to consider.

Definition 9. Suppose $F\colon (\mathrm{N} \to \mathrm{N}) \times \mathrm{N} \rightharpoonup \mathrm{N}$ is $\mathcal{B}$-continuous.

(a) A *locking segment* for F is a (σ, x) for which there is a $y \in \mathrm{N}$ such that for all f with $(f,x) \in (\!(\sigma,x)\!)$, $F(f,x) = y$.

(b) A *minimal locking segment* for F is a locking segment (σ,x) for F such that for each $\tau \subset \sigma$, (τ,x) fails to be a locking segment.

(c) The *induced topology* for F (denoted $\mathcal{I}(F)$) is the topology determined by the subbasis: $\{\,(\!(\sigma,x)\!) \mid (\sigma,x)$ is a minimal locking segment for $F\,\}$. ◇

Definition 10. The *induced topology* for the $\boldsymbol{\varphi}$-program i (denoted $\mathcal{I}_i$) is the topology determined by the subbasis: $\{\,(\!(\mathrm{Use}_i(f,x),x)\!) \mid f\colon (\mathrm{N}\to\mathrm{N}) \to \mathrm{N},\ x \in \mathrm{N}\,\}$. ◇

It is easily seen, for each i with $\boldsymbol{\varphi}_i \in \mathcal{RF}$, that $\mathcal{I}(\boldsymbol{\varphi}_i)$ is a subtopology of $\mathcal{I}_i$ which in turn is a subtopology of $\mathcal{B}$ and that $\mathcal{I}(\boldsymbol{\varphi}_i)$ is *the* smallest subtopology of $\mathcal{B}$ such that $\boldsymbol{\varphi}_i$ is continuous.

Now we have a decision to make. We can take "the run time of $\boldsymbol{\varphi}$-program i is almost everywhere bounded by β" as meaning either (a) $E_{i,\beta}$ is $\mathcal{I}_i$-compact or (b) $E_{i,\beta}$ is $\mathcal{I}(\boldsymbol{\varphi}_i)$-compact. Choice (a) exactly matches our talk about counting computations. But working with choice (a) turns out to be tricky. Part of the problem is that under choice (a) it is very hard to compare computations of programs for the same functional — $\boldsymbol{\varphi}_i = \boldsymbol{\varphi}_j$ does not imply that $\mathcal{I}_i$ and $\mathcal{I}_j$ have much to do with one another. So for reason simplicity, in *this* paper we make choice (b). There are prices to be paid for this choice, but they are generally tolerable. Thus we officially introduce:

Definition 11. We say that the run time of $\boldsymbol{\varphi}$-program i is *almost everywhere bounded by* β if and only if $E_{i,\beta}$ is $\mathcal{I}(\boldsymbol{\varphi}_i)$-compact. $\diamond$

Note: Since $\mathcal{I}(\boldsymbol{\varphi}_i)$ is a subtopology of $\mathcal{I}_i$, any $\mathcal{I}_i$-compact set is also $\mathcal{I}(\boldsymbol{\varphi}_i)$-compact. We shall use this frequently in the following.

As a first check that Definition 11 is reasonable, we note:

Proposition 12. *Suppose i is such that $\boldsymbol{\varphi}_i \in \mathcal{RF}$ and $c \in \mathrm{N}$. Then $\{\, (f,x) \mid \boldsymbol{\Phi}_i(f,x) \le c \,\}$ is $\mathcal{I}(\boldsymbol{\varphi}_i)$-compact.*

Proof. Suppose that $\boldsymbol{\Phi}_i(f,x) \le c$. It follows from our restrictions on OTMs that $|x|$, $|\max(Q_i(f,x))|$, and $|\max\{\, \mathrm{Use}_i(f,x)(y) \mid y \in Q_i(f,x) \,\}|$ are all no greater than c. Clearly then, there are only finitely-many computations of $\boldsymbol{\varphi}$-program i with $\boldsymbol{\Phi}_i(f,x) \le c$. Therefore, $\{\, (f,x) \mid \boldsymbol{\Phi}_i(f,x) \le c \,\}$ is $\mathcal{I}_i$-compact, and hence, $\mathcal{I}(\boldsymbol{\varphi}_i)$-compact. $\square$

§6. Type-2 complexity classes

Now that all the pieces are in place, we can state:

Definition 13. For each $\beta \in \mathrm{WB}$:

$$\mathbf{C}(\beta) \quad =_{\mathrm{def}} \quad \{\, \boldsymbol{\varphi}_i \in \mathcal{RF} \mid i \in \mathrm{N} \;\&\; E_{i,\beta} \text{ is } \mathcal{I}(\boldsymbol{\varphi}_i)\text{-compact} \,\}. \qquad (2)$$

We call $\mathbf{C}(\beta)$ the *complexity class* named by β. $\diamond$

This notion of complexity class is similar its type-1 cousin in many ways. Here is a first illustration.

Proposition 14. *Suppose $F \in \mathbf{C}(\beta)$. Then there is a $\boldsymbol{\varphi}$-program i for F and a $c \in \mathrm{N}$ such that $E_{i,c\cdot\beta} = \emptyset$.*

Proof Sketch. Let p be such that $\varphi_p = F$ and $E_{p,\beta}$ is $\mathcal{I}(F)$-compact. Let $\mathbf{M}$ be the OTM coded by p and let $\mathcal{C}$ be a finite $\mathcal{I}(F)$-cover of $E_{p,\beta}$. If $\mathcal{C} = \emptyset$, then we are done. Suppose $\mathcal{C} \neq \emptyset$ and let $\{\, x_0, \dots, x_k \,\} = \{\, x \mid (\!(\sigma, x)\!) \in \mathcal{C} \,\}$. One can argue that, for each $i \leq k$, there is a finite decision tree T_i as follows.

Each node n of T_i is labeled a $y_n \in \mathrm{N}$. If n is an interior node, this will correspond to the oracle query "$f(y_n) =?$". If n is a terminal node, this will correspond to the output being y_n. Each edge leaving an interior node is labeled a $z \in \mathrm{N}$; this corresponds to z being the answer to the interior node's query. For each node n of T_i, let σ_n be the finite function that corresponds to the set of queries and answers on the path leading to n. We require that (i) if n is a terminal node, then (σ_n, x_i) a locking segment for φ_p and $\varphi_i(\overline{\sigma_n}, x_i) = y_n$; (ii) if n is an interior node, then for all $f \supset \sigma_n$, $\mathbf{M}$ on (f, x_i) queries f on y_n, and (iii) $\{\, (\!(\sigma_n, x_i)\!) \mid n \text{ is a terminal node of } T_i \text{ and } i \leq k \,\}$ covers $E_{p,\beta}$.

Given these T_i's, let $\mathbf{M}'$ be the OTM that, on input (f, x), checks if $x = x_i$ for some $i \leq k$. If not, then $\mathbf{M}'$ acts like $\mathbf{M}$. If so, then $\mathbf{M}'$ follows the decision tree T_i until either (i) it reaches an terminal node n, in which case is outputs y_n and halts or else (ii) $\mathbf{M}'$ reaches an interior node n, and $f(y_n)$ is not the label of any edge leaving n, in which case, $\mathbf{M}'$ acts like $\mathbf{M}$ on input (f, x).

Clearly, $\mathbf{M}'$ computes φ_p. The extra cost of running $\mathbf{M}'$ on (f, x) over running $\mathbf{M}$ is the cost of following the decision tree T_i when $x = x_i$ for some $i \leq k$. Since in following the decision tree T_i simply involves making queries that $\mathbf{M}$ on input (f, x) will have to make anyhow. Hence, with a little careful programming, there is a $c \in \mathrm{N}$ such that $c \cdot \beta$ everywhere bounds the run time of $\mathbf{M}'$. □

Note: In general it is false that if $E_{i,\beta}$ is $\mathcal{I}(\varphi_i)$-compact, then there is a c such that $E_{i,\beta+c} = \emptyset$ — this is part of the price of using the $\mathcal{I}(\varphi_i)$ topology.

As a second illustration of the similarity between type-1 and type-2 complexity classes, we show that a straightforward lift of the proof of Rabin's Theorem (Theorem 2) suffices to obtain a type-2 version of that result.

Theorem 15. *Suppose $\beta \in \mathrm{WB}$. Then there is an 0–1-valued element $F \in \mathcal{RF}$ such that $F \notin \mathbf{C}(\beta)$.*

Proof Sketch. The argument is a direct lift of the one given for Theorem 2 above. In the program for F given in Figure 2, $C_{f,w}$ = programs cancelled on inputs (f, w') with $w' < w$ and $S_{f,w}$ = the candidates for cancellation on input (f, w). A program i is *cancelled* on input (f, w) if and only if $i \in C_{f,w+1} - C_{f,w}$, in which case the construction will guarantee that $\varphi_{i,\beta}(f, w)\Downarrow$, $F(f, w) \neq \varphi_i(f, w)$, and i will never be cancelled again on an input of the form (f, x) with $x > w$.

It is clear from the program that cancellation works as advertised and F is a 0-1 element of $\mathcal{RF}$. To show $F \notin \mathbf{C}(\beta)$ consider an i such that $E_{i,\beta}$ is

```
Input $(f,x)$.
$C_{f,0} \leftarrow \emptyset$.
For $w \leftarrow 0$ to $x$ do:
    $S_{f,w} \leftarrow \{\, k \leq w \mid k \notin C_{f,w}\ \&\ \varphi_{k,\beta}(f,w)\Downarrow \,\}$.
    If $S_{f,w} \neq \emptyset$ then $C_{f,w+1} \leftarrow C_{f,w} \cup \{\, \min(S_{f,w}) \,\}$ else $C_{f.w+1} \leftarrow C_{f,w}$.
If $S_{f,x} = \emptyset$ then return 0 else return $1 \dot{-} \varphi_e(f,x)$, where $e = \min(S_{f,x})$.
```

Figure 2: The program for F

$\mathcal{I}(F)$-compact. Fix $f\colon \mathrm{N} \to \mathrm{N}$ and choose $w_0 \geq i$ so that (a) $\mathbf{\Phi}_{i,\beta}(f,w_0)\Downarrow$ and (b) for all $k < i$ that are ever cancelled on an input of the form (f,x) have been cancelled by input w_0. Hence, either i has been cancelled on a (f,w) with $w < w_0$ or the construction must cancel i on input (f,w_0). In either case $\varphi_i \neq F$. Therefore, $F \notin C(\beta)$. □

So much for similarities, the next two sections demonstrate some marked differences between type-1 and type-2 complexity classes. To keep this paper a reasonable size we shall omit proofs in these next two sections, but the proofs can be found in [Li01].[3]

§7. *Unions of complexity classes*

The type-1 situation. The class of type-1 polynomial-time computable functions is commonly referred to as a complexity class, but it is far from obvious that there is a $t_P \in \mathcal{R}$ that names exactly that class. That there is such a t_P follows from the following quite difficult result, which holds when Φ is an arbitrary complexity measure.

Theorem 16 (The Union Theorem, McCreight and Meyer [MM69, Odi99]). *Suppose that $t\colon \mathrm{N}^2 \to \mathrm{N}$ is computable and nondecreasing in its first argument. Then there is a computable $g\colon \mathrm{N} \to \mathrm{N}$ such that $C(g) = \bigcup_i C(\lambda x \,.\, t(i,x))$.*

This theorem is barely true in the sense that you want the g of the theorem to have any nice properties (e.g., honesty), then you find the result breaks.

The type-2 situation. Since the Union Theorem is fairly delicate, it is no surprise that it is fails to hold in its full strength at type-2. However, the failure is spectacular. Here is an important example of this.

Theorem 17 (Li [Li01]). *The class of type-2 basic feasible functionals fails to be a type-2 complexity class.*

[3]A late draft of this is available as: ftp://ftp.cis.syr.edu/users/royer/CCLthesis.ps.

To obtain some measure of how bad this failure is and to obtain some sufficient conditions on a weak version the union theorem at type-2, we introduce some conditions on type-2 complexity bounds. To keep this paper a reasonable length, we shall not explain these notions beyond their definitions.

DEFINITION 18.

(a) We say that (σ, x) is a *locking fragment* of β (denoted $\beta(\sigma, x)\downarrow$) if and only if for all $\tau \supseteq \sigma$, $\beta(\tau, x) = \beta(\sigma, x)$.

(b) We say that $\ell\colon \mathcal{F} \times \mathrm{N} \to \{0, 1\}$ is a *locking detector* for β if and only if (i) ℓ is computable, (ii) for each f and x, $\lim_{\sigma \to f} \ell(\sigma, x) = 1$, and (iii) for each σ and x, $\ell(\sigma, x) = 1$ implies that $\beta(\sigma, x)\downarrow$.

(c) A *minimal locking fragment* of β is a locking fragment of β such that, for all for all $\tau \subsetneq \sigma$, (τ, x) fails to be a locking fragment of β.

(d) We say that β is *useful* if and only if for every (σ, x), minimal locking fragment of β, we have that, for each $\tau \subseteq \sigma$, $\beta(\tau, x) \geq \|\tau\| + |x| + 2$.[4] $\diamond$

DEFINITION 19. Let $\langle\beta_i\rangle_{i\in\mathrm{N}}$ be a sequence of elements of WB such that the function $\lambda i, \sigma, x\,.\,\beta_i(\sigma, x)$ is computable. We say that:

(a) $\langle\beta_i\rangle_{i\in\mathrm{N}}$ is *ascending* if and only if, for all i, $\beta_i \leq \beta_{i+1}$.

(b) $\langle\beta_i\rangle_{i\in\mathrm{N}}$ is *useful* if and only if each β_i is useful.

(c) $\langle\beta_i\rangle_{i\in\mathrm{N}}$ is *convergent* if and only if, for each f and x, there is a $\sigma_{f,x} \subset f$ such that for all i, $\beta_i(\sigma_{f,x}, x)\downarrow$.

(d) $\langle\beta_i\rangle_{i\in\mathrm{N}}$ *uniformly convergent* if and only if, for all i, x, and σ, if $\beta_i(\sigma, x)\downarrow$, then for all j, $\beta_j(\sigma, x)\downarrow$.

(e) $\langle\beta_i\rangle_{i\in\mathrm{N}}$ *strongly convergent* if and only if $\langle\beta_i\rangle_{i\in\mathrm{N}}$ is uniformly convergent and there is a locking detector for β_0. $\diamond$

THEOREM 20 (LI [Li01]). *There is a ascending, useful, convergent $\langle\beta_i\rangle_{i\in\mathrm{N}}$ such that $\bigcup_i \mathbf{C}(\beta_i)$ is not a complexity class.*

This is a fairly strong non-union result. We conjecture that convergent can be strengthened to uniformly convergent in the previous theorem. By strengthening the hypotheses on the β_i's even more, we can obtain the following weak union theorem. We conjecture that this theorem fails if we require all the complexity bounds to be elements of SB.

THEOREM 21 (THE WEAK TYPE-2 UNION THEOREM, LI [Li01]). *Suppose that $\langle\beta_i\rangle_{i\in\mathrm{N}}$ is ascending, useful, and strongly convergent. Then there is a $\beta \in$ WB such that $\mathbf{C}(\beta) = \bigcup_i \mathbf{C}(\beta_i)$.*

[4] There is a different definition of useful in [Li01] that is more understandable, but requires a bit of back-story.

One nice consequence of this theorem is that type-2 big-O classes *are* complexity classes. We conjecture, however, that the SB version of the following is false.

COROLLARY 22. *Suppose* $\beta \in \mathrm{WB}$. *Let* $\boldsymbol{O}(\beta) = \bigcup_{a,b\in\mathrm{N}} \mathbf{C}(a\cdot\beta + b)$. *Then* $\boldsymbol{O}(\beta)$ *is a complexity class.*

§*8. Gaps and compressions*

The type-1 situation. We know by Rabin's Theorem that for each $t \in \mathcal{R}$, there is a $t' \in \mathcal{R}$ such that $C(t) \subsetneq C(t')$. However, effectively constructing such a t' from a given t turns out to be impossible as shown by the following two theorems. (**N.B.** Constructing such a t' from a *program* for t is easy, but that is not the issue here.)

THEOREM 23 (THE GAP THEOREM, BORODIN [BOR72]). *For each* $r \in \mathcal{R}$, *there is an increasing* $t \in \mathcal{R}$ *such that* $C(t) = C(r \circ t)$, *in fact, there is no* i *with* $t \leq^* \Phi_i \leq^* r \circ t$.

THEOREM 24 (THE OPERATOR GAP THEOREM, CONSTABLE [CON72] AND YOUNG [YOU73]). *For each recursive operator* $\Theta\colon (\mathrm{N}\to\mathrm{N}) \to (\mathrm{N}\to\mathrm{N})$, *there is an increasing* $t \in \mathcal{R}$ *such that* $C(t) = C(\Theta(t))$, *in fact, there is no* i *with* $t \leq^* \Phi_i \leq^* \Theta(t)$.

In both of these theorems, the reason for the gap in which no Φ_i lives is that the t's in question are pathologically dishonest. If we restrict our attention to more sensible names for complexity classes, we obtain the following result that matches our intuitions a bit better.

THEOREM 25 (THE COMPRESSION THEOREM, BLUM [BLU67]). *There is a computable* $r\colon \mathrm{N}^2 \to \mathrm{N}$ *such that for all* i *with* $\varphi_i \in \mathcal{R}$, *we have* $C(\Phi_i) \subsetneq C(\lambda x \centerdot r(x, \Phi_i(x)))$.

The type-2 situation. The fact that the β's are tied to queries imposes a sort of honesty on our time bounds. We thus loose the gap phenomenon at type-2 as shown by:

THEOREM 26 (THE WB INFLATION THEOREM, LI [LI01]). *There is a recursive operator* Θ *such that, for each* $\beta \in \mathrm{WB}$, $\Theta(\beta) \in \mathrm{WB}$ *and* $\mathbf{C}(\beta) \subsetneq \mathbf{C}(\Theta(\beta))$.

We do obtain a gap theorem for unions. But given the wiggly nature of unions, this is not surprising nor is it particularly hard to show.

THEOREM 27 (THE UNION-GAP THEOREM, LI [LI01]). *For each recursive operator* Θ *such that for each* $\beta \in \mathrm{WB}$, $\Theta(\beta) \in \mathrm{WB}$, *there is an ascending* $\langle\beta_i\rangle_{i\in\mathrm{N}}$ *such that* $\bigcup_i \mathbf{C}(\beta_i) = \bigcup_i \mathbf{C}(\Theta(\beta_i))$.

§9. *Conclusion*

General type-2 complexity theory is almost completely unknown territory. In this paper we have blazed one path into this territory. This path is obviously not the only such and it likely is not the best, but we feel that it represents a creditable bit of exploration. In particular we suspect that the failure of the union and gap theorems will be features of any reasonable complexity theory for type-2.

There are obviously many open questions: What happens with the speedup theorems? What happens if we restrict all the β's to SB? What if we change to the unit cost model for OTMs? If we are stuck with naming large classes (e.g., the type-2 basic feasible functions) through unions, what are the general properties of these union classes. (Li [Li01] addresses many of these questions.) Going a little farther, one can ask: How can one extend our work to cover the effectively continuous type-2 functionals? (This requires a careful treatment of computation over partial (e.g., $\mathrm{N} \rightharpoonup \mathrm{N}$) arguments.) How can we extend this work beyond type-2? (Our notion of complexity bound seems amenable to realizer-based definitions of higher-type classes.)

References

[Blu67] M. Blum, *A machine-independent theory of the complexity of recursive functions*, Journal of the Association for Computing Machinery **14** (1967), 322–336.

[Bor72] A. Borodin, *Computational complexity and the existence of complexity gaps*, Journal of the Association for Computing Machinery **19** (1972), 158–174.

[CK90] S. Cook and B. Kapron, *Characterizations of the basic feasible functions of finite type*, Feasible Mathematics: A Mathematical Sciences Institute Workshop, (S. Buss and P. Scott, eds.), Birkhäuser, 1990, pp. 71–95.

[Con72] R. Constable, *The operator gap*, Journal of the Association for Computing Machinery **19** (1972), 175–183.

[Con73] R. Constable, *Type two computational complexity*, Proc. of the Fifth Ann. ACM Symp. on Theory of Computing, 1973, pp. 108–121.

[CU93] S. Cook and A. Urquhart, *Functional interpretations of feasibly constructive arithmetic*, Annals of Pure and Applied Logic **63** (1993), 103–200.

[HS65] J. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Transactions of the American Mathematical Society **117** (1965), 285–306.

[IKR01] R. Irwin, B. Kapron, and J. Royer, *On characterizations of the basic feasible functional, Part I*, Journal of Functional Programming (2001), to appear.

[Kap91] B. Kapron, *Feasible computation in higher types*, Ph.D. thesis, Department of Computer Science, University of Toronto, 1991.

[KC96] B. Kapron and S. Cook, *A new characterization of type 2 feasibility*, SIAM Journal on Computing **25** (1996), 117–132.

[Li01] C.-C. Li, *Type-2 complexity theory*, Ph.D. thesis, Syracuse University, 2001.

[MM69] E. McCreight and A. Meyer, *Classes of computable functions defined by bounds on computation*, Proc. of the First Ann. ACM Symp. on Theory of Computing, 1969, pp. 79–88.

[Odi89] P. Odifreddi, *Classical recursion theory*, North-Holland, 1989.

[Odi99] P. Odifreddi, *Classical recursion theory, volume II*, North-Holland, 1999.

[Rab60] M. Rabin, *Degree of difficulty of computing a function and a partial ordering of the recursive sets*, Report 2, University of Jerusalem, 1960.

[Rog67] H. Rogers, *Theory of recursive functions and effective computability*, McGraw-Hill, 1967, reprinted, MIT Press, 1987.

[Set92] A. Seth, *There is no recursive axiomatization for feasible functionals of type 2*, Seventh Annual IEEE Symposium on Logic in Computer Science, 1992, pp. 286–295.

[Set94] A. Seth, *Complexity theory of higher type functionals*, Ph.D. thesis, University of Bombay, 1994.

[You73] P. Young, *Easy constructions in complexity theory: Gap and speed-up theorems*, Proceedings of the American Mathematical Society **37** (1973), 555–563.

Safe weak minimization revisited (Extended abstract)

Dieter Spreen
Fachbereich Mathematik, Theoretische Informatik
Universität Siegen, 57068 Siegen, Germany
Email: spreen@informatik.uni-siegen.de

1 Introduction

As is well known, unbounded minimization allows the generation of the computable functions from the primitive recursive ones. If one restricts to bounded minimization then the latter class is closed under this operation. By relaxing the minimization condition in such a way that no longer the smallest zero of a given function is asked for, but the smallest argument that is mapped onto an even result, Bellantoni [1, 2] could derive a machine-independent characterization of the class $\square^p$ of functions computable in deterministic polynomial time by querying oracles in the polynomial-time hierarchy. The result is in the style of Cobham's classical characterization of the polytime functions [4]. Bellantoni showed that the class $\square^p$ is the smallest class of functions containing certain basic functions and being closed under substitution, limited recursion on notation and his weakened minimization operator.

Surprisingly, the characterization remains true in a tiered (safe) framework [6, 7, 3], where, in addition to safe composition and safe (predicative) recursion on notation, an unbounded version of relaxed minimization is used that only allows to minimize safe arguments.

The result shows that even in its relaxed form minimization (bounded or unbounded) is still a powerful operation. In a recent paper Danner and Pollett [5] weakened Bellantoni's safe minimization operation again by rst limiting the veri cation that a computed function argument c is minimal to only those numbers d that are less than the length of c and then prescribing which bits of c a further computation may have access to. This operation, called limited safe weak minimization, is necessarily multi-valued. They showed that the smallest class of multifunctions generated from certain initial functions by safe composition, safe recursion on notation and this new operation is exactly the class **NPMV** of partial multifunctions computable in nondeterministic polynomial time.

As already stated by the authors, the de nition of limited safe weak minimization is reminiscent of limited minimization, thus not in the spirit of implicit computational complexity. In this paper we propose a modi ed version of safe weak minimization which remedies this problem. Moreover, we show that the above mentioned characterization of **NPMV** holds true when the new version of safe weak minimization is used.

2 Basic definitions and facts

Inputs to functions are categorized as *normal* or *safe*, the normal ones being written to the left of a semicolon and the safes ones to the right. The variables x, y, z are usually used in normal position, and a, b, c are usually used in safe position. For a function class B the subclass consisting of the functions with only normal arguments is denoted by $\mathrm{Norm}(B)$.

Write $|x|$ for the binary length $\lceil \log_2(x+1) \rceil$ of integer x; the terms "predecessor", "successor" refer to binary notation. If $\bar{x}$ is a vector of n integers, then write $|\bar{x}|$ for the vector $|x_1|, \ldots, |x_n|$ and write $\bar{f}(\bar{x})$ for $f_1(\bar{x}), \ldots, f_m(\bar{x})$.

De nition 2.1 Let B_0 consist of the functions (1)-(5) below:

1. (**Constant**) 0 (a zero-ary function).
2. (**Projections**) $\pi_j^{n,m}(x_1, \ldots, x_n; x_{n+1}, \ldots, x_{n+m}) = x_j$, for $1 \quad j \quad n+m$.
3. (**Successors**) $s_0(;a) = 2a$ and $s_1(;a) = 2a+1$. Write "ai" for $s_i(;a)$, where $i \in \{0,1\}$.
4. (**Predecessor**) p such that $p(;0) = 0$ and $p(;a0) = p(;a1) = a$.
5. (**Conditional**)

$$\mathrm{cond}(;a,b,c) = \begin{cases} b & \text{if } a \bmod 2 = 0, \\ c & \text{otherwise.} \end{cases}$$

De nition 2.2 $[B_0; \mathrm{SC}, \mathrm{SRN}]$ denotes the smallest class of functions containing B_0 and closed under the operations (1) and (2):

1. (**Safe Composition**) Given h, $\bar{r}$ and $\bar{t}$ de ne f by

$$f(\bar{x}; \bar{a}) = h(\bar{r}(\bar{x};); \bar{t}(\bar{x}; \bar{a})).$$

2. (**Safe Recursion on Notation**) Given g and h_0, h_1 de ne f by, for $i \in \{0,1\}$,

$$\begin{aligned} f(0, \bar{x}; \bar{a}) &= g(\bar{x}; \bar{a}) \\ f(yi, \bar{x}; \bar{a}) &= h_i(y, \bar{x}; \bar{a}, f(y, \bar{x}; \bar{a})) \quad \text{for } yi \neq 0. \end{aligned}$$

Let **FP** be the class of all polytime functions. The following result is due to Bellantoni and Cook [1, 3].

Theorem 2.3 **FP** $= \mathrm{Norm}([B_0; \mathrm{SC}, \mathrm{SRN}])$.

This theorem gives strong reasons to consider the polytime functions as the complexity-theoretic analog of the primitive recursive functions. So the question comes up what is the complexity-theoretic analog of the partial recursive functions and can the functions in this class be generated from the polytime functions by applying a suitable minimization operator. This question was the starting point for recent investigations by Danner and Pollett [5].

In his dissertation [1, 2] Bellantoni introduced a safe minimization operator.

De nition 2.4 (**Safe Minimization**) Given h, de ne f by

$$f(\bar{x}; \bar{a}) = \begin{cases} s_1(; \mu b.\, h(\bar{x}; \bar{a}, b) \bmod 2 = 0) & \text{if there is such a } b, \\ 0 & \text{otherwise.} \end{cases}$$

Moreover, he showed that the functions computable on a polynomial-time bounded oracle Turing machine with an oracle in the polynomial-time hierarchy are exactly the functions in $[B_0; \mathrm{SC}, \mathrm{SRN}, \mathrm{SM}]$ that have only normal arguments.

This class is far beyond of what could be considered as a complexity-theoretic analog of the partial recursive functions. In their paper [5] Danner and Pollett introduced two weakenings of safe minimization: safe weak minimization and limited safe weak minimization, and by using the latter operator instead of safe minimization they showed that the collection of functions in the resulting class that have only normal arguments is exactly the class **NPMV** of all partial multifunctions computable in nondeterministic polynomial time, thus answering the above posed question.

A *partial multifunction* is a map $f\colon \mathbb{N}^k \rightharpoonup \mathcal{P}_{\mathrm{fin}}(\mathbb{N})$ for some k, where $\mathcal{P}_{\mathrm{fin}}(\mathbb{N})$ is the collection of all nite subsets of the natural numbers. Alternatively, f can be viewed as a relation on $\mathbb{N}^{k+1}$ satisfying the constraint that for all $\bar{x}$, $\{\, y \mid (\bar{x}, y) \in f \,\}$ is nite. We write $f(\bar{x}) \mapsto y$ when y is a (possible) outcome of f.

De nition 2.5 (**Safe Weak Minimization**) Given a partial multifunction g, f is de ned by

$$f(\bar{x}; \bar{a}) \mapsto b \Leftrightarrow g(\bar{x}; \bar{a}, b) \bmod 2 \mapsto 0 \wedge (\forall c < |b|) g(\bar{x}; \bar{a}, c) \bmod 2 \mapsto 1.$$

If f is de ned by safe weak minimization from g we write $f(\bar{x}; \bar{a}) = \mu^w b.\, g(\bar{x}; \bar{a}, b) \bmod 2 = 0$.

As is shown in the next example, if f is de ned from g by safe weak minimization then for any inputs $\bar{x}$ and $\bar{a}$, $f(\bar{x}; \bar{a})$ may have superexponentially many outputs, even if g is single-valued.

Example 2.6 Set $g(x; a) = \mathrm{cond}(; P(x; b), 1, 0)$, where the function P is de ned by, for $i \in \{0, 1\}$,

$$P(0; b) = b, \qquad P(xi; b) = p(; P(x; b)).$$

$P(x; b)$ takes $|x|$ predecessors of b.

It follows that $g(x; b)$ has value 0, if the $(|x| + 1)$-st low-order bit of b is 1, otherwise its value is 1. Now, let

$$f(x;) = \mu^w b.\, g(x; b) \bmod 2 = 0.$$

Then $f(x;) \mapsto b$, for all numbers b such that

- $|b| \quad 2^{|x|}$ and
- the $(|x| + 1)$-st low-order bit of b is 1.

By de nition, if the $(|x|+1)$-st low-order bit of b is 0 then $g(x; b) = 1$ and hence $f(x;) \not\mapsto b$. Now, suppose that $|b| > 2^{|x|}$ and $g(x; b) \bmod 2 = 0$. As $g(x; 2^{|x|}) \bmod 2 = 0$, it follows that $f(x;) \not\mapsto b$. On the other hand, if $|b| \quad 2^{|x|}$ so that the $(|x| + 1)$-st low-order bit of b is 1 then $f(x;) \mapsto b$, since for d with $d < |b|$ one has that $|d| \quad |x|$ and hence that the $(|x| + 1)$-st low-order bit of b is 0, which means that $g(x; d) = 1$.

Thus, the cardinality of $\{\, b \mid f(x;) \mapsto b \,\}$ is equal to the cardinality of the set of binary sequences of length $2^{|x|} \quad 1$, ie.

$$\| \{\, b \mid f(x;) \mapsto b \,\} \| = 2^{2^{|x|} \quad 1}.$$

Since for any function $h \in \mathbf{NPMV}$ one has that $\| \{ z \mid h(\bar{x}) \mapsto z \} \| \quad 2^{p(|\bar{x}|)}$, for some polynomial p, it follows that the class **NPMV** is not closed under safe weak minimization, which means that this operation is still too strong.

De ne the function $a \,\underline{\mathrm{mod}}\, v = a \bmod 2^{|v|}$. Then $a \,\underline{\mathrm{mod}}\, v$ is the number given by the $|v|$ low-order bits of a. For a sequence $\bar{a} = a_1, \ldots, a_k$, write $\bar{a} \,\underline{\mathrm{mod}}\, v$ for $a_1 \,\underline{\mathrm{mod}}\, v, \ldots, a_k \,\underline{\mathrm{mod}}\, v$.

De nition 2.7 (**Limited Safe Weak Minimization**) Given partial multifunctions g and h, f is de ned by

$$f(\bar{x}; \bar{a}) = (\mu^w b.\, g(\bar{x}; \bar{a}, b) \bmod 2 = 0) \,\underline{\mathrm{mod}}\, h(\bar{x};).$$

Theorem 2.8 **NPMV** $= \mathrm{Norm}([B_0; \mathrm{SC}, \mathrm{SRN}, \mathrm{LSWM}])$.

Here, safe composition and safe recursion on notation have been transferred to the partial multifunction setting in the obvious way.

If f is de ned by limited safe weak minimization from g and h then the application of $\underline{\mathrm{mod}}$ cuts the possibly superexponentially many outputs b of $\mu^w b.\, g(\bar{x}; \bar{a}, b) \bmod 2 = 0$ down to those with $|b| \quad |z|$, for some z with $h(\bar{x};) \mapsto z$, thus to exponentially many. Limited safe weak minimization is reminiscent of limited minimization. As Danner and Pollett remark, by using this operation one enters the "gray area" of implicit computational complexity.

3 Safe minimization on notation

In this section we present a modi cation of safe weak minimization which follows the ideas of implicit computational complexity and show that an analog of Theorem 2.8 holds.[1] The idea is to minimize with respect to the pre x order on binary representations.

For numbers a and b, respectively, let $a_n \ldots a_0$ and $b_m \ldots b_0$ be their binary representations. De ne $a \sqsubseteq b$ if $n \quad m$ and $a_i = b_i$, for all $i \quad n$. Write $a \sqsubset b$ if $a \sqsubseteq b$ and $a \neq b$.

De nition 3.1 A partial multifunction $f(\bar{x}; \bar{a})$ is *consistent* if not both $f(\bar{x}; \bar{a}) \bmod 2 \mapsto 0$ and $f(\bar{x}; \bar{a}) \bmod 2 \mapsto 1$.

De nition 3.2 (**Safe Minimization on Notation**) Given a consistent partial multifunction g, f is de ned by

$$f(\bar{x}; \bar{a}) \mapsto b \Leftrightarrow g(\bar{x}; \bar{a}, b) \bmod 2 \mapsto 0 \wedge (\forall c \sqsubset b) g(\bar{x}; \bar{a}, c) \bmod 2 \mapsto 1$$

Write $f(\bar{x}; \bar{a}) = \mu^n b.\, g(\bar{x}; \bar{a}, b) \bmod 2 = 0$, if f is de ned by safe minimization on notation from g.

Example 3.3 In order to see the e ect of the modi ed quanti cation in the second condition, consider Example 2.6 again. As has been shown,

$$\{ b \mid f(x;) \mapsto b \} = \{ b \mid |x| + 1 \quad |b| \quad 2^{|x|} \} = \{ b \mid 2^{|x|} \quad b < 2^{2^{|x|}} \}.$$

Since the function g is single-valued, it is consistent. De ne

$$f'(x;) = \mu^n b.\, g(x; b) \bmod 2 = 0.$$

[1] This research has been motivated by an earlier version of Danner and Pollett's paper in which only safe weak minimization was studied. It should be noted that the de nitions and results in this section have been obtained independently of what the authors presented in their revised version.

Then $\{ b \mid f'(x;) \mapsto b \}$ is the set of all numbers of minimal length such that the ($|x| + 1$)-st bit is 1, which means that it is the set of all numbers of exactly length $|x| + 1$. Hence

$$\{ b \mid f'(x;) \mapsto b \} = \{ b \mid 2^{|x|} \quad b < 2^{|x|+1} \}.$$

For numbers a and b let $a \preceq b$ if $|a| \quad |b|$. Then $\preceq$ is a preorder. Write $a \prec b$ if $a \preceq b$ and $a \neq b$.

Obviously, $a \sqsubset b$ exactly if $a = b \,\underline{\mathrm{mod}}\, d$, for some number d with $|d| = |a|$. Therefore we have the following lemma which is useful in the derivation of our main result below.

Lemma 3.4 *Let f and g be partial multifunctions and let g be consistent. Then f is obtained by safe minimization on notation from g if and only if for any $\bar{x}$, $\bar{a}$ and b,*

$$f(\bar{x}; \bar{a}) \mapsto b \Leftrightarrow g(\bar{x}; \bar{a}, b) \bmod 2 \mapsto 0 \wedge (\forall c \prec b) g(\bar{x}; \bar{a}, b \,\underline{\mathrm{mod}}\, c) \mapsto 1.$$

Theorem 3.5 $\mathbf{NPMV} = \mathrm{Norm}([B_0; \mathrm{SC}, \mathrm{SRN}, \mathrm{SMN}])$.

The theorem follows from the following propositions.

Proposition 3.6 *Let $f \in \mathbf{NPMV}$. Then there are functions $T(z; \bar{a}, c)$, $\mathrm{res}(z; a)$ and $\mathrm{bnd}(\bar{x};)$ in $[B_0; \mathrm{SC}, \mathrm{SRN}]$ such that for all inputs $\bar{x}$*

$$f(\bar{x}) = \mathrm{res}(\mathrm{bnd}(\bar{x};); \mu^n b.\, T(\mathrm{bnd}(\bar{x};); \bar{x}, b) \bmod 2 = 0).$$

The proposition follows in the usual way by an appropriate coding of Turing machine computations. Note that for any accepting computation there is a smallest accepting sub-computation with the same result.

Corollary 3.7 *Let $f(\bar{x}) \in \mathbf{NPMV}$. Then $f(\bar{x};) \in [B_0; \mathrm{SC}, \mathrm{SRN}, \mathrm{SMN}]$.*

The proof of the converse implication uses a technique of Bellantoni.

De nition 3.8 Let f be a partial multifunction (note that we do not separate the arguments into normal and safe here) and let q be a polynomial.

1. Function $f(\bar{x}, \bar{a})$ is a *polynomial checking function on $\bar{x}$ with threshold q* if for all $\bar{x}$, $\bar{a}$, w and v satisfying $|v| \quad q(|\bar{x}|) + |w|$,

$$f(\bar{x}; \bar{a}) \,\underline{\mathrm{mod}}\, w = f(\bar{x}; \bar{a} \,\underline{\mathrm{mod}}\, v) \,\underline{\mathrm{mod}}\, w.$$

2. Function $f(\bar{x}, \bar{a})$ is *polymax bounded by q on $\bar{x}$* if for all $\bar{x}$, $\bar{a}$ and y with $f(\bar{x}, \bar{a}) \mapsto y$, $|y| \quad q(|\bar{x}|) + \max_i |a_i|$.

Note that the equation in De nition 3.8(1) has to be understood as an equation between sets.

Proposition 3.9 *If $f(\bar{x}; \bar{a})$ is in $[B_0; \mathrm{SC}, \mathrm{SRN}, \mathrm{SMN}]$, then f is a polymax-bounded polynomial checking function on $\bar{x}$.*

Corollary 3.10 *Let $f(\bar{x}; \bar{a})$ be a partial multifunction in $[B_0; \mathrm{SC}, \mathrm{SRN}, \mathrm{SMN}]$. Then $f(\bar{x}, \bar{a})$ is computable in nondeterministic polynomial time.*

The proof proceeds by induction on the derivation of f in $[B_0; \mathrm{SC}, \mathrm{SRN}, \mathrm{SMN}]$. By the preceding proposition the length of any output of f is bounded by a polynomial in the length of the input. This can be used to design polynomial-time algorithms for the computation of f in case that f is obtained by an application of safe recursion on notation or safe minimization on notation, respectively.

Note that the consistency requirement in De nition 3.2 is needed in the proof of Proposition 3.9. As can be seen from the following example, without this condition the class **NPMV** would not be closed under safe minimization on notation, which means that Theorem 3.5 and hence Proposition 3.9 would be false.

Example 3.11 Let $g(x, b) \mapsto 0, 1$, for all $x, b \in \mathbb{N}$. Then $g \in$ **NPMV**, but g is not consistent. Now, de ne f by

$$f(x) \mapsto b \Leftrightarrow g(x, b) \bmod 2 \mapsto 0 \wedge (\forall c \sqsubset b) g(x, c) \bmod 2 \mapsto 1.$$

Then it is readily veri ed that $f(x) \mapsto z$, for all $z \in \mathbb{N}$. Thus, $f \notin$ **NPMV**, since otherwise there exist some polynomial p such that $\| \{ z \mid f(x) \mapsto z \} \| \quad 2^{p(|x|)}$.

References

[1] S. Bellantoni, Predicative Recursion and Computational Complexity, Ph.D. Thesis, University of Toronto, Toronto, 1992.

[2] S. Bellantoni, Predicative recursion and the polytime hierarchy, in: P. Clote and J. Remmel, eds., *Feasible Mathematics II*, 15–29, Birkhauser, Boston, 1995.

[3] S. Bellantoni and S. Cook, A new recursion-theoretic characterization of the polytime functions, *Computational Complexity* 2 (1992) 97–110.

[4] A. Cobham, The intrinsic computational di culty of functions, in: Y. Bar-Hillel, ed., *Logic, Methodology and Philosophy of Science II*, North-Holland, Amsterdam, 1965, 24–30.

[5] N. Danner and C. Pollett, Minimization and NP multifunctions, manuscript, 2001.

[6] D. Leivant, Subrecursion and lambda representation over free algebras (Preliminary summary), in: S. Buss an P. Scott, eds, *Feasible Mathematics*, Birkhauser, Boston, 1990.

[7] D. Leivant, Rami ed recurrence and computational complexity I: Word recurrence and polytime, in: P. Clote and J. Remmel, eds., *Feasible Mathematics II*, 320–343, Birkhauser, Boston, 1995.