TIERED ARITHMETICS

HELMUT SCHWICHTENBERG (MÜNCHEN DE)
STANLEY S. WAINER (LEEDS UK)

Abstract. In his paper "Logics for Termination and Correctness of Functional Programs, II. Logics of Strength PRA" [4] Feferman was concerned with the problem of how to guarantee the feasibility (or at least the subrecursive complexity) of functions definable in certain logical systems. His ideas have influenced much subsequent work, for instance the final chapter of [10]. There, linear two-sorted systems LT(;) (a version of Gödel's T) and LA(;) (a corresponding arithmetical theory) of polynomial-time strength were introduced. Here we extend LT(;) and LA(;) in such a way that some forms of non-linearity are covered as well. This is important when one wants to deal on the proof level with particular algorithms, not only with the functions they compute. Examples are divide-and-conquer approaches as in treesort, and the first of two main sections here gives a detailed analysis of this. The second topic treated heads in a different direction, though again its roots lie in the final chapter of [10]. Instead of just two sorts we consider transfinite ramified sequences of such sorts (or "tiers"). A hierarchy of infinitary arithmetical systems $\mathrm{EA}(I_\alpha)$ is devised, being weak analogues of the iterated inductive definitions underpinning much of Feferman's work. Their strengths turn out to correspond to the levels of the extended Grzegorczyk or Fast-Growing hierarchy. A "pointwise" concept of transfinite induction provides an ordinal measure of strength, but it is a weak (essentially finitistic) notion, related to the Slow-Growing hierarchy.

Keywords: Polynomial time, linear two-sorted arithmetic, program extraction, tiered arithmetic, fast and slow-growing hierarchies, pointwise transfinite induction.

1. Introduction

The principle of numerical induction:

$$A(0) \land \forall_a(A(a) \to A(a+1)) \to A(n)$$

may be viewed as being "impredicative", since establishing that n has property A might entail quantifying over all numbers, in particular n itself, even before it is completely understood (cf. Nelson [8]). "Tiering" is a way to

unravel such impredicativities: one thinks of the input n as existing at a higher level (tier) than A's quantified variables – the output domain. Such tiered or predicative inductions severely restrict the computational strength of an arithmetical theory (cf. Leivant [6]).

For example, suppose we want to build a theory of elementary recursive strength. Then we will have the exponential function $E(n)$ representing 2^n:

$$E(0) := 1, \qquad E(S(n)) := D(E(n)).$$

However, its iteration

$$F(0) := 1, \qquad F(S(n)) := E(F(n))$$

should be avoided. Let us see how a proof of "computational strength" of F could arise. Let $F(n) \simeq a$ denote the graph of F, viewed as an inductively defined relation. We write $F(n)\downarrow$ for $\exists_a(F(n) \simeq a)$. Then a proof of

$$\forall_n F(n)\downarrow$$

should be disallowed. Such a proof would be by induction on n. In the step we would need to prove $\forall_n(F(n)\downarrow \to F(S(n))\downarrow)$. So assume n is given and we have an a such that $F(n) \simeq a$. We need to find b such that $E(F(n)) \simeq b$. Clearly $E(a)$ would be such a b, but here we have substituted the "output" variable a in the recursion (or "input") argument of E. We therefore distinguish input and output variables and argument positions in order to avoid superexponential strength. The underlying idea is that the inputs form a new "tier" lying over the domain of output values. While an input may be fed down to the output level and used as a bound on induction or recursion steps, this is a one-way process - for outputs may not be fed back as inputs.

If we want to even further restrict the computational strength to the subelementary level, we need a linearity restriction. Consider for example the function $B(a, n)$ representing $a + 2^n$:

$$B(a, 0) := S(a), \qquad B(a, S(n)) := B(B(a, n), n).$$

Let $B(a, n) \simeq b$ be the (inductively defined) graph of B, and consider the following proof of

$$\forall_{n,a} B(a, n)\downarrow,$$

by induction on n. In the step we have n and can assume $\forall_a B(a, n)\downarrow$; we need to show $\forall_a B(a, S(n))\downarrow$, i.e., $\exists_c(B(B(a, n), n) \simeq c)$. Given n and a, by induction hypothesis we have b such that $B(a, n) \simeq b$, and applying the induction hypothesis again, we have c such that $B(b, n) \simeq c$. This double use of the induction hypothesis is responsible for the exponential growth,

and hence we use a linearity restriction to stay within the subelementary realm.

2. Representing algorithms in linear two-sorted arithmetic

In this section we define the constructive systems A(;) and LA(;), their intended use being to develop program specification proofs and then term extraction for practical algorithms. The computational strength of A(;) will be elementary recursive (going back to early developments of such theories by Leivant [6], based on the safe / normal discipline of Bellantoni and Cook [1] and earlier Simmons [11]). The subtheory LA(;) will be corresponding theory of polynomial strength, and therefore relevant for the development of feasible programs.

The main contents of this section will be a description of these theories and their basic properties, followed by examples illustrating their use. In order to build such these we first need to define their term structures, which will incorporate higher types.

2.1. The term systems T(;) and LT(;).

We consider *types* built from base types ι by two forms $\rho \hookrightarrow \sigma$ and $\rho \to \sigma$ of arrow types, called input arrow and output arrow. A type is *safe* if it does not involve the input arrow $\hookrightarrow$.

As base types we have the type $\mathbf{B}$ of booleans $\mathtt{tt}$, $\mathtt{ff}$, the (unary) natural numbers $\mathbf{N}$ with constructors 0 and $S\colon \mathbf{N} \to \mathbf{N}$, products $\rho \times \sigma$ with constructor $\times^+\colon \rho \to \sigma \to \rho \times \sigma$ and lists $\mathbf{L}(\rho)$ with constructors $[]$ and $::_\rho$ of type $\rho \to \mathbf{L}(\rho) \to \mathbf{L}(\rho)$. Note that all constructors have safe types.

Variables are typed, and come in two forms, input variables $\bar{x}^\rho$ and output variables x^ρ. Constants are (i) the constructors and (ii) the recursion operators for base types, for instance

$$\mathcal{R}^\tau_{\mathbf{N}}\colon \mathbf{N} \hookrightarrow \tau \to (\mathbf{N} \hookrightarrow \tau \to \tau) \hookrightarrow \tau,$$
$$\mathcal{R}^\tau_{\mathbf{L}(\rho)}\colon \mathbf{L}(\rho) \hookrightarrow \tau \to (\rho \hookrightarrow \mathbf{L}(\rho) \hookrightarrow \tau \to \tau) \hookrightarrow \tau,$$

where the value type τ is required to be safe. This requirement is necessary because without it we could define the iterated exponential function F from the exponential function E via iteration with value type $\mathbf{N} \hookrightarrow \mathbf{N}$. We also have (iii) the cases operators for base types, for instance

$$\mathcal{C}^\tau_{\mathbf{N}}\colon \mathbf{N} \to \tau \to (\mathbf{N} \hookrightarrow \tau) \to \tau,$$
$$\mathcal{C}^\tau_{\mathbf{L}(\rho)}\colon \mathbf{L}(\rho) \to \tau \to (\rho \hookrightarrow \mathbf{L}(\rho) \hookrightarrow \tau) \to \tau,$$
$$\mathcal{C}^\tau_{\rho\times\sigma}\colon \rho \times \sigma \to (\rho \hookrightarrow \sigma \hookrightarrow \tau) \to \tau,$$

where again the value type τ is required to be safe.

Remark. Recursion and cases operators are provided for all base types. However, with arbitrary base types, we may have more than one recursive call. If – as in 2.1 – we are to develop a theory based on linear ideas, we must disallow recursion operators with multiple recursive calls, since this would spoil the whole approach.

T(;)-*terms* are built from variables and the constants above by introduction and elimination rules for the two type forms $\rho \hookrightarrow \sigma$ and $\rho \to \sigma$:

$$\begin{aligned}
&\bar{x}^\rho \mid x^\rho \mid C^\rho \text{ (constant)} \mid \\
&(\lambda_{\bar{x}^\rho} r^\sigma)^{\rho\hookrightarrow\sigma} \mid (r^{\rho\hookrightarrow\sigma} s^\rho)^\sigma \ (s \text{ an input term}) \mid \\
&(\lambda_{x^\rho} r^\sigma)^{\rho\to\sigma} \mid (r^{\rho\to\sigma} s^\rho)^\sigma.
\end{aligned}$$

A term s is an *input term* if all its free variables are input variables, or else s is of higher type and all its higher type free variables are input variables.

Remark. The restriction for $(r^{\rho\hookrightarrow\sigma} s^\rho)^\sigma$ is more liberal here than in [10]: we now allow output variables of base type in case s is of higher type. This change does not affect the estimates ensuring elementary complexity. We also changed the type of step terms in recursion and cases operators, for instance $\rho \hookrightarrow \mathbf{L}(\rho) \hookrightarrow \tau \to \tau$ instead of $\rho \to \mathbf{L}(\rho) \to \tau \to \tau$ in $\mathcal{R}^\tau_{\mathbf{L}(\rho)}$. This makes is easier to construct step terms as lambda-abstractions, since now the abstracted variables corresponding to parts of the recursion argument are input variables and hence usable to build input terms. These changes do not affect the complexity estimates.

LT(;)-*terms* are built from variables and the constants above by introduction and elimination rules for the two type forms $\rho \hookrightarrow \sigma$ and $\rho \to \sigma$, but now with an additional linearity restriction:

$$\begin{aligned}
&\bar{x}^\rho \mid x^\rho \mid C^\rho \text{ (constant)} \mid \\
&(\lambda_{\bar{x}^\rho} r^\sigma)^{\rho\hookrightarrow\sigma} \mid (r^{\rho\hookrightarrow\sigma} s^\rho)^\sigma \ (s \text{ an input term}) \mid \\
&(\lambda_{x^\rho} r^\sigma)^{\rho\to\sigma} \mid (r^{\rho\to\sigma} s^\rho)^\sigma \ (\text{higher type output variables in } r, s \text{ distinct}, \\
&\qquad\qquad r \text{ does not start with a cases operator } \mathcal{C}^\tau_\iota) \mid \\
&\mathcal{C}^\tau_\iota t \vec{r} \qquad\qquad (\text{higher type output variables in } \mathrm{FV}(t) \text{ not in } \vec{r})
\end{aligned}$$

with as many r_i as there are constructors of ι. The notion of an input term is the same as above. The restriction on output variables in the formation of $r^{\rho\to\sigma} s$ or $\mathcal{C}^\tau_\iota t \vec{r}$ ensures that every higher type output variable can occur at most once in a given LT(;)-term, except in the alternatives of a cases operator.

2.2. **The theories** A(;) **and** LA(;)**.** We consider *formulas* built from atomic formulas by (i) the two forms $A \hookrightarrow B$ and $A \to B$ of implication, called input and output implication, and (ii) universal quantification either $\forall_{\bar{x}} A$ over an input variable $\bar{x}$ or $\forall_x A$ over an output variable x. *Atomic formulas* are either terms of the type $\mathbf{B}$ of booleans viewed as propositions or else inductively defined predicates – possibly with parameters – applied to argument terms. We view $\exists_{\bar{x}} A$ and $\exists_x A$ as atomic formulas, more precisely as (nullary) inductive predicates with predicates $\{\, \bar{x} \mid A \,\}$ or $\{\, x \mid A \,\}$ as parameter.

In this section proofs are in minimal logic, in natural deduction style. It is convenient to represent them as proof terms, as in Table 1. For quantification on input variables $\bar{x}$ we have similar rules, and also for the input implication $\hookrightarrow$. Assumption variables come in two forms, input ones $\bar{u}^A$ and output ones u^A. Axioms are the introduction and elimination axioms for $\exists$

$$\exists^+_{\{\, x \mid A \,\}} \colon \forall_x(A \to \exists_x A), \qquad \exists^-_{\{\, x \mid A(x) \,\},P} \colon \exists_x A(x) \to \forall_{\bar{x}}(A(\bar{x}) \hookrightarrow P) \to P$$

and similarly for input variables $\bar{x}$. For every base type we have its induction and cases axioms, for instance

$$\mathrm{Ind}_{\bar{n},P} \colon \forall_{\bar{n}}(P(0) \to \forall_{\bar{n}}(P(\bar{n}) \to P(S(\bar{n}))) \hookrightarrow P(\bar{n}^{\mathbf{N}})),$$
$$\mathrm{Ind}_{\bar{l},P} \colon \forall_{\bar{l}}(P([]) \to \forall_{\bar{x},\bar{l}}((P(\bar{l}) \to P(\bar{x} :: \bar{l})) \hookrightarrow P(\bar{l}^{\mathbf{L}(\rho)}))$$

and

$$\mathrm{Cases}_{n,P} \colon \forall_n(P(0) \to \forall_{\bar{n}} P(S(\bar{n})) \to P(n^{\mathbf{N}})),$$
$$\mathrm{Cases}_{l,P} \colon \forall_l(P([]) \to \forall_{\bar{x},\bar{l}} P(\bar{x} :: \bar{l}) \to P(l^{\mathbf{L}(\rho)})).$$

We call these *raw* proof terms. Note that when ignoring the annotations of implications and variables we obtain proofs terms in ordinary arithmetic. The raw proof terms need to be restricted to make up the theories A(;) and LA(;). To formulate these restrictions it is easiest to refer to the *extracted term* $\mathrm{et}(M)$ of a proof term M, which we introduce first. This requires some preparations.

Computational content in proofs arises from computationally relevant (c.r.) atomic formulas; in our setting the only ones are $\exists_{\bar{x}} A$ and $\exists_x A$. There are also non-computational (n.c.) atomic formulas, like equalities. Following Kolmogorov [5] we assign to every formula A an object $\tau(A)$, which is a type or the "nulltype" symbol $\circ$. The definition can be conveniently written if we extend the use of $\rho \hookrightarrow \sigma$, $\rho \to \sigma$ and $\rho \times \sigma$ to the nulltype symbol $\circ$:

$$(\rho \hookrightarrow \circ) := \circ, \quad (\circ \hookrightarrow \sigma) := \sigma, \quad (\circ \hookrightarrow \circ) := \circ \quad \text{and similarly for } \to,$$
$$(\rho \times \circ) := \rho, \quad (\circ \times \sigma) := \sigma, \quad (\circ \times \circ) := \circ.$$

Derivation	Term
$u\colon A$	u^A
$\dfrac{\begin{array}{c}[u\colon A]\\ \mid M\\ B\end{array}}{A\to B}\to^+ u$	$(\lambda_{u^A} M^B)^{A\to B}$
$\dfrac{\begin{array}{cc}\mid M & \mid N\\ A\to B & A\end{array}}{B}\to^-$	$(M^{A\to B}N^A)^B$
$\dfrac{\begin{array}{c}\mid M\\ A\end{array}}{\forall_x A}\forall^+ x$ (with var.cond.)	$(\lambda_x M^A)^{\forall_x A}$ (with var.cond.)
$\dfrac{\begin{array}{cc}\mid M & \\ \forall_x A(x) & r\end{array}}{A(r)}\forall^-$	$(M^{\forall_x A(x)} r)^{A(r)}$

TABLE 1. Derivation terms for $\to$ and $\forall$

With this understanding of $\rho\hookrightarrow\sigma$, $\rho\to\sigma$ and $\rho\times\sigma$ we can simply define

$$\begin{aligned}
&\tau(A) := \circ \quad \text{if } A \text{ is an n.c. atomic formula,}\\
&\tau(\exists_{\bar{x}^\rho} A) := \tau(\exists_{x^\rho} A) := \rho\times\tau(A),\\
&\tau(A\hookrightarrow B) := (\tau(A)\hookrightarrow\tau(B)),\\
&\tau(A\to B) := (\tau(A)\to\tau(B)),\\
&\tau(\forall_{\bar{x}^\rho} A) := (\rho\hookrightarrow\tau(A)),\\
&\tau(\forall_{x^\rho} A) := (\rho\to\tau(A)).
\end{aligned}$$

We introduce a special "nullterm" symbol ε to be used as a "realizer" for n.c. formulas, and extend term application to the nullterm symbol by

$$\varepsilon t := \varepsilon, \quad t\varepsilon := t, \quad \varepsilon\varepsilon := \varepsilon.$$

Now we can define the extracted term $\mathrm{et}(M)$ of a proof term M deriving A. It is relative to a fixed assignment of input variables $\bar{x}_{\bar{u}}$ of type $\tau(A)$ to input assumption variables $\bar{u}^A$, and similarly output variables x_u of type $\tau(A)$ to output assumption variables u^A. If A is n.c., then $\mathrm{et}(M) := \varepsilon$, else

$$\begin{aligned}
&\mathrm{et}(\bar{u}^A) && := \bar{x}_{\bar{u}}^{\tau(A)},\\
&\mathrm{et}(u^A) && := x_u^{\tau(A)},\\
&\mathrm{et}((\lambda_{\bar{u}^A} M)^{A\hookrightarrow B}) && := \lambda_{\bar{x}_{\bar{u}}^{\tau(A)}}\mathrm{et}(M),\\
&\mathrm{et}((\lambda_{u^A} M)^{A\to B}) && := \lambda_{x_u^{\tau(A)}}\mathrm{et}(M),\\
&\mathrm{et}(M^{A\hookrightarrow B}N) := \mathrm{et}(M^{A\to B}N) && := \mathrm{et}(M)\mathrm{et}(N),\\
&\mathrm{et}((\lambda_{\tilde{x}^\rho} M)^{\forall_{\tilde{x}}A}) && := \lambda_{\tilde{x}^\rho}\mathrm{et}(M),\\
&\mathrm{et}(M^{\forall_{\tilde{x}}A} r) && := \mathrm{et}(M)r,
\end{aligned}$$

with $\tilde{x}$ input or output variable. We also need to define extracted terms for the axioms, i.e., $\exists^+$, $\exists^-$ and for every base type its induction and cases axioms. The extracted terms are

$$\mathrm{et}(\exists^+_{\{x^\rho \mid A\}}) := \times^+ \qquad \text{of type } \rho \to \tau(A) \to \rho \times \tau(A)$$

$$\mathrm{et}(\exists^-_{\{x^\rho \mid A\},P}) := \mathcal{C}_{\rho\times\tau(A)} \text{ of type } \rho \times \tau(A) \to (\rho \hookrightarrow \tau(A) \hookrightarrow \tau(P)) \to \tau(P)$$

and for the induction and cases axioms the corresponding recursion and cases operators.

Now finally we are ready to define the theories A(;) and LA(;): a raw proof term M is in A(;) (or LA(;)) if $\mathrm{et}(M)$ is a term in T(;) (or LT(;)).

2.3. **Treesort.** In this section we extend LT(;) and LA(;) in such a way that some forms of non-linearity are covered as well. This is important when one wants to deal on the proof level with particular algorithms, not only with the functions they compute. Examples are divide-and-conquer approaches like in treesort. The method requires two recursive calls and hence is not covered by the linear setup in LT(;) and LA(;). However, one can show that the number of conversion steps in the parse-dag computation model still is a polynomial in the length of the list. Generally, one needs to extend LT(;) and LA(;) by constants defined by computation rules meeting certain criteria.

For the formulation of the treesort algorithm we use the base type $\mathbf{T}$ (branch labelled binary trees) with a nullary constructor $\diamond$ and a ternary constructor C of type $\mathbf{N} \to \mathbf{T} \to \mathbf{T} \to \mathbf{T}$. The treesort algorithm is given by the defined constants

$$\begin{aligned}
\mathrm{TreeSort}(l) &= \mathrm{Flatten}(\mathrm{MakeTree}(l)),\\
\mathrm{MakeTree}([]) &= \diamond,\\
\mathrm{MakeTree}(a :: l) &= \mathrm{Insert}(a, \mathrm{MakeTree}(l)),\\
\mathrm{Insert}(a, \diamond) &= C_a(\diamond, \diamond),\\
\mathrm{Insert}(a, C_b(u, v)) &= \begin{cases} C_b(\mathrm{Insert}(a, u), v) & \text{if } a \le b\\ C_b(u, \mathrm{Insert}(a, v)) & \text{otherwise,}\end{cases}\\
\mathrm{Flatten}(\diamond) &= [],\\
\mathrm{Flatten}(C_b(u, v)) &= \mathrm{Flatten}(u) * (b :: \mathrm{Flatten}(v))
\end{aligned}$$

where $*$ denotes the Append-function. Notice that the second defining equation of Flatten has two recursive calls. Therefore this "divide-and-conquer" algorithm is not covered by the treatment in [10]: the linearity restriction is violated. The point of the present section is to show how this problem can be overcome, by giving the non-linear Flatten-function a special treatment w.r.t. our parse dag computation model, which we describe next.

Let LT(;) + Flatten be the extension of LT(;) by the defined constant Flatten of type $\mathbf{T} \hookrightarrow \mathbf{L}(\mathbf{N})$. To obtain a polynomial upper bound on the time complexity of functions definable in LT(;) + Flatten, we need a careful analysis of the normalization process. Our time measurement is with respect to a computation model that fits well to the lambda-terms we have to work with, and is also close to actual computation.

We compute with terms represented as dags (directed acyclic graphs) where only nodes for terms of base type can have in-degree greater than one. Each graph is required to be connected and have a unique root (i.e., node with in-degree zero). Nodes can be (i) terminal nodes labelled by a variable or constant, (ii) abstraction nodes with one successor, labelled with a (typed input or output) variable and a pointer to the successor node, or (iii) application nodes with two successors, labelled with pointers to them. A *parse dag* is required to represent a parse tree for a term, i.e., the types must fit and all other conditions above on the formation of terms must be satisfied.

The *size* $\|d\|$ of a parse dag d is the number of nodes in it. A parse dag is *conformal* if (i) every node with in-degree greater than 1 is of base type, and (ii) every maximal path to a bound variable x passes through the same binding λ_x-node. A parse dag is *h-affine* if every higher type variable occurs at most once in the dag, except in the alternatives of a cases operator. We identify a parse dag with the term it represents.

In our computation model the following steps require one time unit.

(a) Creation of a node given its label and pointers to its successor nodes.
(b) Deletion of a node.
(c) Given a pointer to an interior node, to obtain a pointer to one of its successor nodes.
(d) Test on the type and the label of a node, and on the variable or constant in case the node is terminal.

We will estimate the number of steps it takes to reduce a term t to its normal form $\mathrm{nf}(t)$. For simplicity we fix an order of reduction, by requiring that the leftmost innermost redex is converted first. Let $\#t$ denote the total number of such reduction steps.

Lemma 2.1. *Let l be a numeral of type $\mathbf{L}(\mathbf{N})$. Then*

$$\#(l * l') = O(|l|).$$

Proof. One easily proves $\#(l * l') \leq N \cdot (|l| + 1)$ by induction on $|l|$, for an appropriate N. □

To estimate $\#\mathrm{Flatten}(u)$ we use a size function for numerals u of type $\mathbf{T}$:

$$\| \diamond \| := 0,$$
$$\|C_n(u, v)\| := 2\|u\| + \|v\| + 3.$$

Lemma 2.2. *Let u be a numeral of type $\mathbf{T}$. Then*

$$\#\mathrm{Flatten}(u) = O(\|u\|).$$

Proof. We prove $\#\mathrm{Flatten}(u) \leq N(\|u\|+1)$ by induction on $\|u\|$, for an appropriate N, and only deal with the second defining equation of Flatten, which involves two recursive calls. Consider the parse dag for $\mathrm{Flatten}(C_n(u, v))$. We can assume that it takes $\leq N$ steps to transform it into a the parse dag for $\mathrm{Flatten}(u) * (a :: \mathrm{Flatten}(v))$. Then

$$\#\mathrm{Flatten}(C_n(u, v)) \leq N + \#\mathrm{Flatten}(u) + \#\mathrm{Flatten}(v) + N(\|u\| + 1)$$

using $\#(l * l') \leq N(|l| + 1)$ (by Lemma 2.1) and $|\mathrm{Flatten}(u)| \leq \|u\|$. Hence by induction hypothesis

$$\#\mathrm{Flatten}(C_a(u, v)) \leq N + N(\|u\| + 1) + N(\|v\| + 1) + N(\|u\| + 1)$$

$$= N(2\|u\| + \|v\| + 4)$$
$$= N(\|C_a(u,v)\| + 1). \qquad \square$$

We show that all functions definable in LT(;) + Flatten are polynomial-time computable, by adapting the argument in [10] to the presence of defined constants, here the constant Flatten. Call a term $\mathcal{RD}$-free if it contains neither recursion constants $\mathcal{R}$ nor Flatten. A term is called *simple* if it contains no higher type input variables. Obviously simple terms are closed under reductions, taking of subterms, and applications. Every simple term is h-affine, due to the (almost) linearity of higher type output variables.

As in [10, p.416-418] we have

Lemma 2.3 (Simplicity)**.** *Let t be a base type term whose free variables are of base type. Then* $\mathrm{nf}(t)$ *contains no higher type input variables.*

Lemma 2.4 (Sharing normalization)**.** *Let t be an $\mathcal{RD}$-free simple term. Then a parse dag for* $\mathrm{nf}(t)$*, of size at most $\|t\|$, can be computed from t in time $O(\|t\|^2)$.*

Corollary 2.5 (Base normalization)**.** *Let t be a closed $\mathcal{RD}$-free simple term of type $\mathbf{N}$ or $\mathbf{L}(\mathbf{N})$. Then* $\mathrm{nf}(t)$ *can be computed from t in time $O(\|t\|^2)$, and $\|\mathrm{nf}(t)\| \le \|t\|$.*

Lemma 2.6 ($\mathcal{RD}$-elimination)**.** *Let $t(\vec{x}\,)$ be a simple term of safe type. There is a polynomial P_t such that: if $\vec{r}$ are safe type $\mathcal{RD}$-free closed simple terms and the free variables of $t(\vec{r}\,)$ are output variables, then in time $P_t(\|\vec{r}\,\|)$ one can compute an $\mathcal{RD}$-free simple term* $\mathrm{rdf}(t;\vec{x};\vec{r}\,)$ *such that $t(\vec{r}\,) \to^*$* $\mathrm{rdf}(t;\vec{x};\vec{r}\,)$.

Proof. By induction on $\|t\|$, as in [10, p.418-20]. We only have to add a case for the defined constant Flatten.

Case Flatten. Then t is of the form Flatten(l), because the term has safe type. Since l is an input term, all free variables of l are input variables – they must be in $\vec{x}$ since free variables of $t(\vec{r}\,)$ are output variables. Therefore $l(\vec{r}\,)$ is closed, implying $\mathrm{nf}(l(\vec{r}\,))$ is a list. One obtains $\mathrm{rdf}(l;\vec{x};\vec{r}\,)$ in time $P_l(\|\vec{r}\,\|)$ by the induction hypothesis. Then by base normalization one obtains the lists $\hat{l} := \mathrm{nf}(\mathrm{rdf}(l;\vec{x};\vec{r}\,))$ in a further polynomial time. Now Lemma 2.2 implies the claim. $\square$

Theorem 2.7 (Normalization)**.** *Let t be a closed term in* LT(;) + Flatten *of type $\mathbf{N} \twoheadrightarrow \ldots \mathbf{N} \twoheadrightarrow \mathbf{N}$ ($\twoheadrightarrow \in \{\hookrightarrow, \to\}$). Then t denotes a poly-time function.*

Proof. One must find a polynomial Q_t such that for all $\mathcal{RD}$-free simple closed terms $\vec{n}$ of type $\mathbf{N}$ one can compute $\mathrm{nf}(t\vec{n}\,)$ in time $Q_t(\|\vec{n}\,\|)$. Let $\vec{x}$ be new

variables of type **N**. The normal form of $t\vec{x}$ is computed in an amount of time that may be large, but it is still only a constant with respect to $\vec{n}$. By the simplicity lemma $\mathrm{nf}(t\vec{x}\,)$ is simple. By $\mathcal{RD}$-elimination one reduces to an $\mathcal{RD}$-free simple term $\mathrm{rdf}(\mathrm{nf}(t\vec{x}\,);\vec{x}\,;\vec{n}\,)$ in time $P_t(\|\vec{n}\,\|)$. Since the running time bounds the size of the produced term, $\|\mathrm{rdf}(\mathrm{nf}(t\vec{x}\,);\vec{x}\,;\vec{n}\,)\| \le P_t(\|\vec{n}\,\|)$. By sharing normalization one can compute

$$\mathrm{nf}(t\vec{n}\,) = \mathrm{nf}(\mathrm{rdf}(\mathrm{nf}(t\vec{x}\,);\vec{x}\,;\vec{n}\,))$$

in time $O(P_t(\|\vec{n}\,\|)^2)$, so for Q_t one can take some constant multiple of $P_t(\|\vec{n}\,\|)^2$. □

2.4. **Treesort in** LA(;) + Flatten**.** We have seen that polynomial-time algorithms can be implemented as extracted terms of appropriate proofs in LA(;). This is developed in detail in [10, Section 8.4]. Here we describe that and how the same mechanism works when both LT(;) and LA(;) contain constants (like Flatten) defined by equations involving multiple recursive calls. As an example we treat the treesort algorithm in LA(;).

A tree u is called *sorted* if the list $\mathrm{Flatten}(u)$ is sorted. We recursively define a function I inserting an element a into a tree u in such a way that, if u is sorted, then so is the result of the insertion:

$$I(a,\diamond) := C_a(\diamond,\diamond),$$
$$I(a, C_b(u,v)) := \begin{cases} C_b(I(a,u),v) & \text{if } a \le b, \\ C_b(u, I(a,v)) & \text{if } b < a \end{cases}$$

and, using I, a function S sorting a list l into a tree:

$$S([]) := \diamond, \qquad S(a :: l) := I(a, S(l)).$$

We represent these functions by inductive definitions of their graphs. Thus, writing $I(a,u,u')$ to denote $I(a,u) = u'$ and similarly, $S(l,u)$ for $S(l) = u$, we have the following clauses:

$$I(a,\diamond,C_a(\diamond,\diamond)),$$
$$a \le b \to I(a,u,u') \to I(a, C_b(u,v), C_b(u',v)),$$
$$b < a \to I(a,v,v') \to I(a, C_b(u,v), C_b(u,v')),$$

and

$$S([],\diamond),$$
$$S(l,u) \to I(a,u,u') \to S(a :: l, u').$$

As an auxiliary function we use $\mathrm{tl}_i(l)$, which is the tail of the list l of length i, if $i < |l|$, and l otherwise. Its recursion equations are

$$\mathrm{tl}_i([]) := [], \quad \mathrm{tl}_i(a :: l) := \begin{cases} \mathrm{tl}_i(l) & \text{if } i \le |l| \\ a :: l & \text{else.} \end{cases}$$

We will need some easy properties of S and tl:

$$\begin{aligned}
&S([a], C_a(\diamond, \diamond)),\\
&S(l, C_b(u, v)) \to a \le b \to I(a, u, u') \to S(a :: l, C_b(u', v)),\\
&S(l, C_b(u, v)) \to b < a \to I(a, v, v') \to S(a :: l, C_b(u, v')),\\
&i \le |l| \to \mathrm{tl}_i(a :: l) = \mathrm{tl}_i(l),\\
&\mathrm{tl}_{|l|}(l) = l, \qquad \mathrm{tl}_0(l) = [].
\end{aligned}$$

We would like to derive $\exists_u S(l, u)$ in LA(;). However, we shall not be able to do this. All we can achieve is $|l| \le n \to \exists_u S(l, u)$, with n an input parameter.

Lemma 2.8 (Tree insertion)**.** $\forall_{a,n,u}(|u| \le n \to \exists_{u'} I(a, u, u'))$.

Proof. We fix a and use induction on n. In the base case we can take $u' := C_a(\diamond, \diamond)$. In the step assume the induction hypothesis and $|w| \le n+1$. If $|w| \le n$ use the induction hypothesis. Now assume $n < |w|$. Then w is of the form $C_b(u, v)$. If $a \le b$ pick u' by induction hypothesis and take $C_b(u', v)$. If $b < a$ pick v' by induction hypothesis and take $C_b(u, v')$. □

Lemma 2.9 (Treesort)**.** $\forall_{l,n,m}(m \le n \to \exists_u S(\mathrm{tl}_{\min(m,|l|)}(l), u))$.

Proof. We fix l, n and use induction on m. In the base case we can take $u := \diamond$, using $\mathrm{tl}_0(l) = []$. In the step case assume the induction hypothesis and $m + 1 \le n$. If $|l| \le m$ we are done by the induction hypothesis. If $m < |l|$ we must show $\exists_u S(\mathrm{tl}_{m+1}(l), u)$. Now $\mathrm{tl}_{m+1}(l) = a :: \mathrm{tl}_m(l)$ with $a := \mathrm{hd}(\mathrm{tl}_{m+1}(l))$, since $m < |l|$. If $m = 0$ take $u := C_a(\diamond, \diamond)$. If $0 < m$, by induction hypothesis we have w with $S(\mathrm{tl}_m(l), w)$, and w is of the form $C_b(u, v)$. If $a \le b$, pick u' by Lemma 2.8 such that $I(a, u, u')$. Take $S(a :: \mathrm{tl}_m(l), C_b(u', v))$. If $b < a$, pick v' by Lemma 2.8 such that $I(a, v, v')$. Take $S(a :: \mathrm{tl}_m(l), C_b(u, v'))$. □

We have formalized these proofs in Minlog and extracted their computational content, i.e., LT(;)-terms. For Lemma 2.8 we obtain a term involving the recursion operator $\mathcal{R}^{\tau}_{\mathbf{N}} \colon \mathbf{N} \hookrightarrow \tau \to (\mathbf{N} \hookrightarrow \tau \to \tau) \to \tau$ with $\tau := \mathbf{T} \to \mathbf{T}$. This term represents the function f of type $\mathbf{N} \to \mathbf{N} \hookrightarrow \mathbf{T} \to \mathbf{T}$ defined by

$$f(a, 0, u) := C_a(\diamond, \diamond),$$

$$f(a,n+1,u) := \begin{cases} f(a,n,u) & \text{if } |u| \le n, \\ C_{\mathrm{Lb}(u)}(f(a,n,L(u)),R(u)) & \text{if } n < |u| \text{ and } a \le \mathrm{Lb}(u), \\ C_{\mathrm{Lb}(u)}(L(u),f(a,n,R(u))) & \text{if } n < |u| \text{ and } \mathrm{Lb}(u) < a \end{cases}$$

with $\mathrm{Lb}(u), L(u), R(u)$ label and left and right subtree of $u \neq \diamond$. For Lemma 2.9 we obtain the function g of type $\mathbf{L}(\mathbf{N}) \to \mathbf{N} \hookrightarrow \mathbf{N} \hookrightarrow \mathbf{T}$ with

$$g(l,n,0) := \diamond, \qquad g(l,n,m+1) :=$$
$$\begin{cases} u & \text{if } |l| \le m, \\ C_{\mathrm{hd}(\mathrm{tl}_1(l))}(\diamond,\diamond), & \text{if } 0 = m < |l|, \\ C_{\mathrm{Lb}(u)}(f(a,m,L(u)),R(u)) & \text{if } 0 < m < |l| \text{ and } a \le \mathrm{Lb}(u) \\ C_{\mathrm{Lb}(u)}(L(u),f(a,m,R(u))) & \text{if } 0 < m < |l| \text{ and } \mathrm{Lb}(u) < a \end{cases}$$

where $u := g(l,n,m)$ and $a := \mathrm{hd}(\mathrm{tl}_{m+1}(l))$.

Specializing Lemma 2.9 to l, n, n we obtain an LA(;)-derivation of

$$|l| \le n \to \exists_u S(l,u). \tag{1}$$

Let $\bar{S}(l,l')$ express that l' is multiset-equal to l and sorted. One easily proves $S(l,u) \to \bar{S}(l,\mathrm{Flatten}(u))$ and thus gets an LA(;) + Flatten-derivation of

$$|l| \le n \to \exists_{l'} \bar{S}(l,l'). \tag{2}$$

From the specialization (1) we get the function h of type $\mathbf{L}(\mathbf{N}) \to \mathbf{N} \hookrightarrow \mathbf{T}$ with $h(l,n) := g(l,n,n)$. For (2) we finally obtain an LT(;) + Flatten-term representing the function $\bar{h}$ of type $\mathbf{L}(\mathbf{N}) \to \mathbf{N} \hookrightarrow \mathbf{L}(\mathbf{N})$ with $\bar{h}(l,n) := \mathrm{Flatten}(h(l,n))$.

3. Transfinitely iterated tiering

This section investigates the effect of adding to an arithmetical base theory a transfinite hierarchy of number-theoretic tiers $\{I_\alpha\}$, indexed by countable "tree ordinals" α (that is, ordinals with fixed, chosen fundamental sequences $\{\lambda_i\}_{i\in N}$ assigned to limits λ). I_0 will be the "output" domain and I_1 the first level of "inputs", controlling the lengths of inductions on formulas of level 0. Thus from the previous section and chapter 8 of [10] one notes already that a (non-linear) arithmetical theory incorporating $I_0 + I_1$ will have computational strength = elementary. Now let us see what may be gained by adding successively higher levels of input, and what conditions should be placed on those levels. Initially we will work quite informally, but will make things more precise later.

A version of the fast-growing hierarchy is:

$$F_0(n) = n+1\ ; \quad F_{\beta+1}(n) = F_\beta^{2^n}(n)\ ; \quad F_\lambda(n) = F_{\lambda_n}(n)\ .$$

The lesson is that each new tier will allow a new level of iteration/induction, so let us assume that we can prove $F_\beta : I_\beta \to I_\beta$, that is

$$\forall_x(I_\beta(x) \to \exists_y(I_\beta(y) \wedge F_\beta(x) \simeq y))\ .$$

Then the formula $A(z) \equiv F_\beta^{2^z}{:}I_\beta \to I_\beta$ is inductive, and so by *tiered induction*: $\forall_z(I_{\beta+1}(z) \to A(z))$. The tiering also entails $I_{\beta+1}(z) \to I_\beta(z)$ and therefore by definition of $F_{\beta+1}(z)$,

$$\forall_z(I_{\beta+1}(z) \to \exists_y(I_\beta(y) \wedge F_{\beta+1}(z) \simeq y))\ .$$

Now a further principle of tiering is that the level of a "computed" variable y may be "lifted" as high as the lowest level of non-zero input on which it depends (a form of Σ^0_1-reflection rule as used in Cantini [3]). Therefore the $I_\beta(y)$ may be lifted to $I_{\beta+1}(y)$ so that

$$\forall_z(I_{\beta+1}(z) \to \exists_y(I_{\beta+1}(y) \wedge F_{\beta+1}(z) \simeq y))\ .$$

Hence $F_{\beta+1} : I_{\beta+1} \to I_{\beta+1}$.

At limits λ we want to show $F_\lambda : I_\lambda \to I_\lambda$. Assume inductively that it holds already at each stage λ_x of the fundamental sequence to λ, thus

$$\forall_z(I_{\lambda_x}(z) \to \exists_y(I_{\lambda_x}(y) \wedge F_{\lambda_x}(z) \simeq y))\ .$$

The condition placed on limit tiers must now be $I_\lambda(x) \to I_{\lambda_x}(x)$ for then, by the definition of $F_\lambda(x)$:

$$I_\lambda(x) \to \exists_y(I_{\lambda_x}(y) \wedge F_\lambda(x) \simeq y)\ .$$

Again, since the $\exists_y$ only depends upon x at level λ, the $I_{\lambda_x}(y)$ may be lifted to $I_\lambda(y)$ and this completes the limit step.

To summarize, the main principles are (i) tiered induction, (ii) lifting, and (iii) diagonal: $I_\lambda(x) \to I_{\lambda_x}(x)$. These are formalized below, in a hierarchy of infinitary arithmetics $\mathrm{EA}(I_\alpha)$ whose computational strengths correspond exactly to the levels $\mathcal{E}^\alpha$ of the fast-growing, extended Grzegorczyk hierarchy.

3.1. **The Infinitary Systems** $\mathrm{EA}(I_\alpha)$**.** Given α, the infinitary system $\mathrm{EA}(I_\alpha)$ derives Tait-style sequents with numerical input declarations:

$$n_1{:}I_{\beta_1}, \ldots n_j{:}I_{\beta_j} \vdash^\gamma \Gamma \quad \text{abbreviated} \quad \vec{n}{:}\vec{I} \vdash^\gamma \Gamma$$

where $\alpha \succeq \beta_1 \succ \ldots \succ \beta_j$. Γ will be a finite set of closed formulas in the language of arithmetic augmented by elementary term constructors for

sequence-coding and suitably large ordinal notations, and a new binary "input predicate" $I_b(m)$ representing $m{:}I_\beta$ when β is the tree ordinal denoted by the notation b, and 0 if b is not a notation. We will write $I_\beta(m)$ for $I_b(m)$, but no confusion should arise. The set Γ is said to be of "level" less than ξ (written $\mathrm{lev}(\Gamma) \prec \xi$) if every I_β occurring has $\beta \prec \xi$. An important convention will be that a declaration $n_j{:}I_{\beta_j}$ where $n_j = 0$ will be suppressed (i.e. assumed and not explicitly stated). Of the declared inputs, only finitely-many will be non-zero. An obvious principle is that $n{:}I_\beta \vdash^\gamma A$ means $\vdash^\gamma I_\beta(n) \to A$ but the declarations $n_j{:}I_{\beta_j}$ to the left of $\vdash^\gamma$ will be kept distinct from the formulas in Γ.

The ordinal bounds γ on the heights of derivations will come from a certain initial segment of tree-ordinals (depending on α) to be specified later. There will be an associated elementary recursive notation-system allowing computation of e.g. successors, predecessors, and limit-projection $(\lambda, n) \mapsto \lambda_n$. It will be assumed that the ordering $\prec$ on tree ordinals will be the union of orderings $\prec_n$ where, for each n, the $\prec_n$ is the transitive closure of $\delta \prec_n \delta + 1$ and $\lambda_n \prec_n \lambda$ for limits λ. Most "standard" notation systems satisfy $\delta \prec_n \gamma \to \delta + 1 \preceq_{n+1} \gamma$ and this too will be a standing assumption here. The "n-descending chain" from γ is

$$0 \prec 1 \prec \ldots \prec \delta \prec \delta + 1 \prec \ldots \prec \lambda_n \prec \lambda \prec \ldots \prec \gamma$$

and this will therefore be contained in the $n+1$-descending chain. Alternative notation for $\gamma' + 1 \preceq_n \gamma$ is $\gamma' \in \gamma[n]$ – see part 2 of [10].

Logic rules. To ensure appropriate levels of stratification, the ordinals γ' bounding the premises of all rules below, bear the following relationship to the ordinal bounds γ assigned to their conclusions: $\gamma' + 1 \preceq_n \gamma$ where $n{:}I_\beta$ is a declared input at a level *higher* than the levels of all formulas in the premises and conclusion. In the case where there is no higher level, that is I_α already appears in Γ, then rules may be applied, but only under the constraint $\gamma = \gamma' + 1$.

The Axioms are $\vec{n}{:}\vec{I} \vdash^\gamma \Gamma$ where the set Γ contains a true atom (e.g. an equation or inequation between closed terms, or $s \neq s', t \neq t', \bar{I}_s(t), I_{s'}(t')$).

The Cut rule, with cut formula C, is

$$\frac{\vec{n}{:}\vec{I} \vdash^{\gamma'} \Gamma, \neg C \qquad \vec{n}{:}\vec{I} \vdash^{\gamma''} \Gamma, C}{\vec{n}{:}\vec{I} \vdash^{\gamma} \Gamma} .$$

The $\exists$-rules are:

$$\frac{\vec{n}{:}\vec{I} \vdash_c^{\gamma'} I_\beta(m) \qquad \vec{n}{:}\vec{I} \vdash^{\gamma''} A(m), \Gamma}{\vec{n}{:}\vec{I} \vdash^{\gamma} \exists_x(I_\beta(x) \wedge A(x)), \Gamma} .$$

Here the left-hand premise "computes" witness m according to the computation rules given below.

The $\forall$-rules are versions of the ω-rule:

$$\frac{\dots \max(n,m){:}I_\beta, \dots \vdash^{\gamma'} A(m), \Gamma \ \text{ for every m in N}}{\dots n{:}I_\beta, \dots \vdash^{\gamma} \forall_x(I_\beta(x) \to A(x)), \Gamma}$$

but note the fixed ordinal bound γ' on the premises, which does not vary with m. This helps to keep the theory "weak".

The $\vee, \wedge$ rules are unsurprising and we don't list them.

The final logic rule allows interaction with computation in the form:

$$\frac{\dots n{:}I_\beta \vdash_c^{\gamma'} I_\beta(m) \qquad \dots m{:}I_\beta, \dots \vdash^{\gamma''} \Gamma}{\dots n{:}I_\beta, \dots \vdash^{\gamma} \Gamma} \ .$$

Computation rules. The same conditions on tree-ordinal bounds γ apply.

The Computational Axioms are $\vec{n}{:}\vec{I} \vdash_c^{\gamma} I_{\beta'}(\ell), \Gamma$ provided there is $n{:}I_\beta$ occurring in the declaration such that $\ell \le n+1$ and $\beta' \preceq_n \beta$. Thus by the $(\forall)$-rule, with any γ, $\vdash^{\gamma+1} \forall_x(I_\beta(x) \to I_{\beta'}(x))$ provided $\beta' \preceq_n \beta$ for every n. Also, with zero declaration, $\vdash^{\gamma+n} I_\beta(n) \to I_{\beta'}(\ell)$ since $\vdash^{\gamma+n-1} I_\beta(n)$, and so I_β is inductive and is contained in $I_{\beta'}$ whenever $\beta' \prec_n \beta$ for all n.

The Lifting Rule, from $I_{\beta'}$ to I_β when $\beta' \prec_n \beta$, is:

$$\frac{\dots n{:}I_\beta \vdash_c^{\gamma} I_{\beta'}(m)}{\dots n{:}I_\beta \vdash_c^{\gamma} I_\beta(m)}$$

recalling that, in the declaration, the blank after $n{:}I_\beta$ means zeros.

The Computation Rules (call-by-value) are:

$$\frac{\dots n{:}I_\beta \vdash_c^{\gamma'} I_\beta(m) \qquad \dots m{:}I_\beta \vdash_c^{\gamma''} I_\beta(\ell)}{\dots n{:}I_\beta \vdash_c^{\gamma} I_\beta(\ell)} \ .$$

Alternative ordinal assignment. Alternatively, in each rule above one could simply take $\gamma = \max(\gamma_0, \gamma_1)+1$ or $= \gamma'+1$. But then, in order to make use of the $\forall$-rule, which requires a fixed bound on all premises, one needs to add an Accumulation Rule, as in Buchholz [2]: from $\vec{n}{:}\vec{I} \vdash^{\gamma'} \Gamma$ derive $\vec{n}{:}\vec{I} \vdash^{\gamma} \Gamma$ provided $\gamma'+1 \preceq_n \gamma$, where n is declared at a level greater than the level of Γ. (This is also a suitably modified version of Mints' Repetition Rule [7].)

Basic lemmas.

Lemma 3.1 (Tiered Induction)**.** *If the level of A is $\prec \beta$ and $\beta' \prec \beta$ and a appropriately measures the "size" of the formula A, so that $\vdash^a A, \neg A$, then*

$$\vdash^{a+2\omega+1} A(0) \wedge \forall_z(I_{\beta'}(z) \to A(z) \to A(z+1)) \to \forall_x(I_\beta(x) \to A(x)) \,.$$

Proof. By repeatedly applying the $(\wedge)$ and $(\exists)$ rules, using $\vdash^a A, \neg A$, one obtains for each m,

$$m{:}I_\beta \vdash^{a+2m+1} A(0) \wedge \forall_z(I_{\beta'}(z) \to A(z) \to A(z+1)) \to A(m) \,.$$

Then $m : I_\beta \vdash^{a+2\omega} A(0) \wedge \forall_z(I_{\beta'}(z) \to A(z) \to A(z+1)) \to A(m)$ because for the tree ordinal ω we take the successor function as its "standard" fundamental sequence and so $m+1 \prec_m \omega$ and hence $a + 2m + 2 \prec_m a + 2\omega$. This holds for every m, so the result follows by applying the $(\forall)$-rule. □

Lemma 3.2 (Bounding)**.** *Let $\{f_\beta(g)\}$ be the following functional version of the fast-growing hierarchy:*

$$\begin{array}{lcl} f_0(g)(n;m) & = & m+1 \\ f_{\beta+1}(g)(n;m) & = & f_\beta(g)(\max(n,m);-)^{2^{g(\max(n,m))}}(m) \\ f_\lambda(g)(n;m) & = & f_{\lambda_n}(g)(n;m)\,. \end{array}$$

Let $G_\gamma(n)$ be the slow-growing function, giving the size of $\{\gamma' : \gamma'+1 \preceq_n \gamma\}$.

Then if $\vec{n}{:}\vec{I} \vdash_c^\gamma I_\beta(m)$ by the computation rules alone, we have:

$$m \leq f_\beta(g)(n;-)^{2^{g(n)}}(\bar{m})$$

where $g = G_\gamma$, $\bar{m} = \max \vec{n}$ and n is the maximum of all declared inputs at levels $\succ \beta$.

Proof. Proceed by induction on β with a nested induction on γ. Let $g' = G_{\gamma'}$ and $g = G_\gamma$ and note that if $\gamma'+1 \preceq_n \gamma$ then $g'(n) < g(n)$.

If $\vec{n}{:}\vec{I} \vdash_c^\gamma I_\beta(m)$ comes about by a computational axiom then $m \leq \bar{m}+1 = f_0(g)(n;\bar{m})$ and the result is immediate.

If it arises by Lifting from $\vec{n}{:}\vec{I} \vdash_c^\gamma I_{\beta'}(m)$ where $\beta' \prec_n \beta$, then inductively one may assume that

$$m \leq f_{\beta'}(g)(\bar{m};-)^{2^{g(\bar{m})}}(\bar{m}) = f_{\beta'+1}(g)(n;\bar{m}) \,.$$

Now since $\beta'+1 \preceq_n \beta \to \beta'+1 \preceq_{n'} \beta$ when $n \leq n'$, it follows that

$$m \leq f_{\beta'+1}(g)(n;\bar{m}) \leq f_\beta(g)(n;\bar{m}) \leq f_\beta(g)(n;-)^{2^{g(n)}}(\bar{m}) \,.$$

Suppose the given derivation comes about by the Computation Rule from premises $\vec{n}{:}\vec{I} \vdash_c^{\gamma'} I_\beta(\ell)$ and $\vec{n}{:}\vec{I}, \ell{:}I_\beta \vdash_c^{\gamma''} I_\beta(m)$. Note that in this

case both $\gamma' + 1$ and $\gamma'' + 1$ are $\preceq_n \gamma$ so $g'(n), g''(n) < g(n)$. Then the induction hypothesis gives $m \leq f_\beta(g'')(n;-)^{2^{g''(n)}}(\max(\bar{m}, \ell))$ and also $\ell \leq f_\beta(g')(n;-)^{2^{g'(n)}}(\bar{m})$. Composing, and at the same time increasing $f_\beta(g')$ and $f_\beta(g'')$ to $f_\beta(g)$,

$$m \leq f_\beta(g)(n;-)^{2^{g'(n)}+2^{g''(n)}}(\bar{m}) \leq f_\beta(g)(n;-)^{2^{g(n)}}(\bar{m})$$

as required. □

Lemma 3.3 (Cut elimination)**.** *(i) Suppose* $\vec{n}{:}\vec{I} \vdash^\gamma \Gamma, \neg C$ *and* $\vec{n}{:}\vec{I} \vdash^\delta \Gamma, C$*, both with cut-rank (maximum size of cut formulas)* $\leq r$*. Suppose also that* C *is either an atom, or a disjunction* $D_0 \vee D_1$ *or of existential form* $\exists_x(I_j(x) \wedge D(x))$ *with* D *of size* r *(the "size" of input predicates is defined to be zero). Then* $\vec{n}{:}\vec{I} \vdash^{\gamma+\delta} \Gamma$ *again with cut-rank* r*.*
(ii) Hence if $\vec{n}{:}\vec{I} \vdash^\gamma \Gamma$ *with cut-rank* $r + 1$ *then* $\vec{n}{:}\vec{I} \vdash^{\omega^\gamma} \Gamma$ *with cut-rank* $\leq r$ *and (repeating this)* $\vec{n}{:}\vec{I} \vdash^{\gamma^*} \Gamma$ *with cut-rank* 0*, where* $\gamma^* = \exp^{r+1}_\omega(\gamma)$*.*

Proof. The proofs are fairly standard. □

Note on Σ^0_1 reflection. The ($\exists$) and "lifting" rules combine to derive the following version of Σ^0_1-reflection:

Suppose one has a cut-free derivation of $n{:}I_\alpha \vdash^\gamma \Gamma$ *where* Γ *is a set of* Σ^0_1 *formulas of level* $\beta' \prec_n \beta \prec_n \alpha$*. Then* $n{:}I_\alpha \vdash^\gamma \Gamma'$ *where* Γ' *results from* Γ *by lifting(some or all) existential quantifiers to level* β*.*

The proof is by induction on γ. Briefly, suppose the premises of the last $\exists$-rule are $n{:}I_\alpha \vdash^{\gamma'} I_{\beta'}(m)$ and $n{:}I_\alpha \vdash^{\gamma''} \Gamma, B(m)$ where Γ contains the formula $\exists_x(I_{\beta'}(x) \wedge B(x))$. Then by the induction hypothesis, $n{:}I_\alpha \vdash^{\gamma''} \Gamma', B'(m)$, and by lifting, $n{:}I_\alpha \vdash^{\gamma'} I_\beta(m)$. Then $n{:}I_\alpha \vdash^\gamma \Gamma', \exists_x(I_\beta(x) \wedge B'(x))$ by reapplying the $\exists$-rule.

3.2. **The Computational Strength of** $\mathrm{EA}(I_\alpha)$**.** To illustrate, we now fix attention on the segment $\alpha \prec \varepsilon_0$ and choose the "standard" notation system for it, based, say, on Cantor normal forms with base ω. The $\mathrm{EA}(I_\alpha)$'s thus provide a "tiering" of Peano Arithmetic.

The heights γ of derivations allowed in $\mathrm{EA}(I_\alpha)$ was previously left open, but now we need to be specific. Thus, henceforth, the heights γ of derivations in $\mathrm{EA}(I_\alpha)$ will also be restricted to $\gamma \prec \varepsilon_0$, allowing $\mathrm{EA}(I_\alpha)$-derivations to be closed under cut elimination.

Theorem 3.4. *The provably recursive functions of* $\mathrm{EA}(I_\alpha)$ *are exactly those functions elementary recursive in* F_α *where* F *is the version of the fast-growing hierarchy defined earlier:*

$$F_0(n) = n+1\ ;\ \ F_{\delta+1}(n) = F_\delta^{2^n}(n)\ ;\ \ F_\lambda(n) = F_{\lambda_n}(n)\ .$$

Proof. $\mathrm{EA}(I_\alpha)$ was devised in the first place, precisely in order to allow derivation of $F_\beta : I_\beta \to I_\beta$ for each $\beta \preceq \alpha$. Furthermore, examination of that argument (in the introduction to this section) would show that the height of this derivation is (of the order) $\omega \cdot \alpha + 2$. The Computation rule will then allow finite compositions of these functions to be formed and derived in $\mathrm{EA}(I_\alpha)$. Thus if f is elementary in F_α (i.e. computable in time bounded by some finite iterate of F_α) then there is an elementary relation $R(n,m)$ such that $f(n)$ may be computed by finding the least m satisfying $R(n,m)$, and furthermore this m is $\leq F_\alpha^k(n)$ for a fixed k. To show that f is provably recursive in $\mathrm{EA}(I_\alpha)$ it is therefore only necessary to prove $n{:}I_\alpha \vdash^\gamma \exists_y(I_\alpha(y) \wedge R(x,y))$ with γ independent of n. But because m is bounded by $F_\alpha^k(n)$ and this provably exists in $\mathrm{EA}(I_\alpha)$, we have $n{:}I_\alpha \vdash_c I_\alpha(m)$ with height independent of n, and also $n{:}I_\alpha \vdash R(n,m)$ since this entails just the checking of bounded quantifiers. Application of the $(\exists)$-rule then gives $n{:}I_\alpha \vdash^\gamma \exists_y(I_\alpha(y) \wedge R(x,y))$.

Conversely, suppose f is provably recursive in $\mathrm{EA}(I_\alpha)$. This means there is an inductively-given or elementary relation R such that $R(n,m) \to f(n) \simeq (m)_0$ holds, and at some level I_β, $\vdash^\gamma \forall_x(I_\alpha(x) \to \exists_y(I_\beta(y) \wedge R(x,y)))$. By inversion, $n{:}I_\alpha \vdash^\gamma \exists_y(I_\beta(y) \wedge R(n,y))$ for every n. This has a cut-free derivation with height $\gamma^* = \exp_\omega^k(\gamma)$ for some fixed k. Therefore (inverting the $\exists_y$ several times if necessary) for each n the correct value m satisfying $R(n,m)$ is such that $n{:}I_\alpha \vdash_c I_\beta(m)$ with height $\preceq_n \gamma^*$. By the Bounding Lemma, $m \leq f_\beta(g)(n;-)^{2^{g(n)}}(n)$ where $g = G_{\gamma^*}$. This g is elementary, because $G_{\gamma^*}(n)$ has the effect of replacing each ω in the Cantor normal form of γ^* by n. Therefore $g(n)$ is bounded by a fixed finite iterate of $F_1(n) = n + 2^n$. It is now not difficult to see, by induction on β, that $f_\beta(g)(n;-) \leq F_\beta(g(\max n, -) + \ell)$, and hence $m \leq F_{\beta+1}(g(n) + \ell)$ for some fixed ℓ. This bound is, as a function of n, elementary in F_α, and so the function f, being given by bounded search (find the least m less than the bound, such that $R(n,m)$) is also elementary in F_α. □

3.3. **Weak, Pointwise Transfinite Induction.** A basic version of transfinite induction up to γ is

$$A(0) \wedge \forall_\delta(A(\delta) \to A(\delta+1)) \wedge \forall_\lambda(\forall_i A(\lambda_i) \to A(\lambda)) \to A(\gamma)\ .$$

Weak, pointwise-at-x transfinite induction up to γ is the following principle:

$$A(0) \wedge \forall_\delta(A(\delta) \to A(\delta+1)) \wedge \forall_\lambda(A(\lambda_x) \to A(\lambda)) \to A(\gamma)$$

where x is a numerical input variable. We denote this principle $\mathrm{PTI}(x, \gamma, A)$ and write $\mathrm{PTI}(x, \gamma)$ for the schema.

Using this, we can immediately prove, with only a small amount of basic coding apparatus, that the x-descending sequence from γ exists. That is

$$\exists_s D(s, x, \gamma)$$

where $D(s, x, \gamma)$ is the bounded formula saying that s is the sequence number of ordinal notations such that $(s)_0 = 0$ and $(s)_{lh(s)-1} = \gamma$ and for each $i < lh(s) - 1$ either $(s)_{i+1}$ is a limit λ, in which case $(s)_i = \lambda_x$, or $(s)_{i+1}$ is a successor $\delta + 1$, in which case $(s)_i = \delta$.

Thus $\exists_s D(s, x, \gamma)$ expresses the pointwise-at-x well-foundedness of γ, and we often abbreviate it as $\mathrm{PWF}(x, \gamma)$. The contrast between this Σ^0_1 notion and full Π^1_1 well-foundedness is stark, but even here there are interesting analogies to be drawn. Whereas the natural subrecursive hierarchies of proof-theoretic bounding functions are "fast" growing in the classical case, they are "slow" growing in the pointwise case. For detailed comparisons between the two, see [10], and Weiermann [12]. Schmerl [9] was the first to formulate such weak, pointwise induction schemes in the context of Peano Arithmetic.

Definition 3.5. The functions L_x and G_x are defined as follows:
$L_x(\gamma) = a$ iff $\exists_s(D(s, x, \gamma) \wedge a = lh(s) - 1)$
$G_x(\gamma) = a$ iff $\exists_s(D(s, x, \gamma) \wedge a = \#(s))$
where $\#(s)$ is the number of successors in the descending sequence s.

Lemma 3.6. *L_x and G_x satisfy the following recursive definitions:*
$L_x(0) = 0$, $L_x(\delta+1) = L_x(\delta) + 1$, $L_x(\lambda) = L_x(\lambda_x) + 1$.
$G_x(0) = 0$, $G_x(\delta+1) = G_x(\delta) + 1$, $G_x(\lambda) = G_x(\lambda_x)$.

These functions, being given "pointwise-at-x", are alternative versions of the slow growing hierarchy, and they are both provably defined as immediate consequences of pointwise well-foundedness. They each have their uses, though we favour G_x since, for each x, it more readily collapses the arithmetic of tree ordinals down onto ordinary arithmetic. Thus writing x as the subscript and γ as the argument (instead of the other way around) is often

a more appropriate notation. We use both, depending on context. Under the assumption $\delta \prec_n \gamma \to \delta + 1 \preceq_{n+1} \gamma$ it immediately follows that

$$G_n(\gamma) \leq L_n(\gamma) \leq G_{n+1}(\gamma) \ .$$

Of course, even to call $\mathrm{PTI}(x, \gamma)$ a *transfinite* induction principle requires a stretch of the imagination, because it is really just a collection of finitary inductions indexed by x and uniformized by γ. The following lemma brings this out more clearly. The levels at which inputs and quantifiers are declared will, for the time being, be suppressed.

Lemma 3.7. *In any arithmetical theory containing the basic coding apparatus,* $\mathrm{PTI}(x, \gamma)$ *implies Numerical Induction up to* $G_x(\gamma)$*, and conversely, Numerical Induction up to* $L_x(\gamma)$ *implies* $\mathrm{PTI}(x, \gamma)$*.*

More precisely, given any formula $F(a)$*, let* $A(\delta) \equiv \forall a \leq G_x(\delta).F(a)$ *where* $\forall a \leq G_x(\delta).F(a)$ *stands for* $\exists_b(G_x(\delta) = b \wedge \forall a \leq b.F(a))$*. Then one may prove (with* x, γ *declared at a level higher than that of* $F(a)$ *and* $A(\delta)$*)*

$$\mathrm{PTI}(x, \gamma, A) \to (\ F(0) \wedge \forall_b(F(b) \to F(b+1)) \to \forall a \leq G_x(\gamma).F(a)\ ) \ .$$

Conversely, given any formula $A(\delta)$ *let* $F(b) \equiv \forall \delta \preceq_x \gamma(L_x(\delta) = b \to A(\delta))$ *where* $\delta \preceq_x \gamma$ *means* $\exists_s(D(s, x, \gamma) \wedge \exists i < lh(s)((s)_i = \delta))$*. Then*

$$(\ F(0) \wedge \forall_b(F(b) \to F(b+1)) \to \forall a \leq L_x(\gamma).F(a)\ ) \to \mathrm{PTI}(x, \gamma, A) \ .$$

Proof. We argue informally. For the first part, it is only necessary to show that the progressiveness of F implies $A(0)$ and $\forall_\delta(A(\delta) \to A(\delta + 1))$ and $A(\lambda_x) \to A(\lambda)$ for limits λ. But $F(0)$ immediately implies $A(0)$. If $\forall_b(F(b) \to F(b+1))$ then $\forall a \leq G_x(\delta).F(a) \to \forall a \leq G_x(\delta+1).F(a)$ which gives $\forall_\delta(A(\delta) \to A(\delta+1))$. The limit case $A(\lambda_x) \to A(\lambda)$ is immediate since $G_x(\lambda_x) = G_x(\lambda)$. Therefore $\mathrm{PTI}(x, \gamma, A)$ gives $A(\gamma) \equiv \forall a \leq G_x(\gamma).F(a)$.

For the converse, assume A is progressive, i.e. $A(0)$ and $\forall_\delta(A(\delta) \to A(\delta+1))$ and $A(\lambda_x) \to A(\lambda)$ at limits λ. Then one easily proves $F(0)$ and for any b, $F(b) \to F(b+1)$. For assume $F(b)$. Then if $\delta \preceq_x \gamma$ and $L_x(\delta) = b + 1$, δ is either a successor or a limit and its immediate predecessor in the $\preceq_x$-sequence, call it δ', satisfies $L_x(\delta') = b$. Therefore $A(\delta')$ holds and, by the progressiveness of A one immediately gets $A(\delta)$. Hence $F(b+1)$, and so by numerical induction up to $L_x(\gamma)$ we then have $F(L_x(\gamma))$ and hence $A(\gamma)$. This implies $\mathrm{PTI}(x, \gamma, A)$. □

The motto is: "In a theory of predicative, or tiered, numerical induction, $G_\gamma\downarrow$ witnesses the provability of pointwise transfinite induction up to γ."

Definition 3.8. Extend G_x to the third number-class by taking large sups to small sups. Thus: $G_x(0) = 0$; $G_x(\delta + 1) = G_x(\delta) + 1$; $G_x(\lambda) = G_x(\lambda_x)$ at small limits λ, and at large limits, $G_x(\mathrm{SUP}_\xi\ \lambda_\xi) = \sup_i G_x(\lambda_i)$.

Note in particular, $G_x(\Omega) = \omega$.

Definition 3.9. For each α in the third number-class, define the function φ_α from countable tree ordinals to countable tree ordinals:

$$\varphi_\alpha(\beta) \;=\; \begin{cases} \beta + 1 & \text{if } \alpha = 0 \\ \varphi^{2^\beta}_{\alpha-1}(\beta) & \text{if } \alpha \text{ is a successor} \\ \sup_i \varphi_{\alpha_i}(\beta) & \text{if } \alpha \text{ is a small limit} \\ \varphi_{\alpha_\beta}(\beta) & \text{if } \alpha \text{ is a large limit.} \end{cases}$$

Lemma 3.10 (Collapsing)**.** *Provided each large limit $\lambda \preceq \alpha$ satisfies the condition $G_x(\lambda_\xi) = G_x(\lambda)_{G_x(\xi)}$, we have:*

$$G_x(\varphi_\alpha(\beta)) \;=\; F_{G_x(\alpha)}(G_x(\beta))\ .$$

Proof. As in chapter 5 of [10]. □

Theorem 3.11. *For each $\alpha \prec \varepsilon_0$, let $\bar\alpha$ be the ordinal in the third number-class obtained by replacing ω by Ω throughout its Cantor normal form. Then $\varphi_{\bar\alpha+1}(\omega)$ is the supremum of the ordinals γ for which $\mathrm{EA}(I_\alpha)$ proves pointwise transfinite induction up to γ.*

Proof. Pointwise transfinite induction up to $\gamma = \varphi_{\bar\alpha+1}(\omega)$ cannot be proven in $\mathrm{EA}(I_\alpha)$ because, by Collapsing, $G_\gamma(n) = F_{G_n(\bar\alpha+1)}(n) = F_{\alpha+1}(n)$ and this is not elementary in F_α. Hence Numerical Induction up to G_γ cannot be proven in $\mathrm{EA}(I_\alpha)$. On the other hand, $\gamma = \sup_i \gamma_i$ where every $\gamma_i = \varphi^{2^i}_{\bar\alpha}(\omega)$. But pointwise transfinite induction up to each γ_i is provable in $\mathrm{EA}(I_\alpha)$ because G_{γ_i} is a finite iteration of F_α and therefore elementary in F_α. □

The φ functions used here are not the Bachmann-Veblen functions ϕ, but are closely related. Thus $\bigcup_{\alpha\prec\varepsilon_0} \mathrm{EA}(I_\alpha)$ is a tiered version of PA^∞ and its provable pointwise transfinite inductions hold up to all ordinals below the Bachmann-Howard $\varphi_{\varepsilon\Omega+1}(\omega)$.

References

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

[2] W. Buchholz. An independence result for Π^1_1–CA+BI. *Annals of Pure and Applied Logic*, 33(2):131–155, 1987.

[3] A. Cantini. Polytime, combinatory logic and positive safe induction. *Archive for Mathematical Logic*, 41(2):169–189, 2002.

[4] S. Feferman. Logics for termination and correctness of functional programs, II. logics of strength PRA. In P. Aczel, H. Simmons, and S. Wainer, editors, *Proof Theory. A selection of papers from the Leeds Proof Theory Programme 1990*, pages 195–225. Cambridge University Press, 1992.

[5] A. N. Kolmogorov. Zur Deutung der intuitionistischen Logik. *Math. Zeitschr.*, 35:58–65, 1932.

[6] D. Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly–time. In P. Clote and J. Remmel, editors, *Feasible Mathematics II*, pages 320–343. Birkhäuser, Boston, 1995.

[7] G. Mints. Finite investigations of transfinite derivations. *Journal of Soviet Mathematics*, 10:548–596, 1978. Translated from: *Zap. Nauchn. Semin. LOMI* 49 (1975).

[8] E. Nelson. *Predicative Arithmetic*. Princeton University Press, 1986.

[9] U. Schmerl. A proof theoretical fine structure in systems of ramified analysis. *Archiv für Mathematische Logik und Grundlagenforschung*, 22:167–186, 1982.

[10] H. Schwichtenberg and S. S. Wainer. *Proofs and Computations*. Perspectives in Logic. Association for Symbolic Logic and Cambridge University Press, 2012.

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

[12] A. Weiermann. What makes a (pointwise) subrecursive hierarchy slow growing? In S. Cooper and J. Truss, editors, *Sets and Proofs: Logic Colloquium '97*, volume 258 of *London Mathematical Society Lecture Notes*, pages 403–423. Cambridge University Press, 1999.